MAC/65™

A Language For
Your ATARI® Computer

™

Precision Software Tools

a reference manual for

M A C / 6 5

a Macro Assembler and Editor program for
use with 6502-based computers built by
Atari, Incorporated

Rev 1.2

PREFACE

MAC/65 is a logical upgrade from the OSS product EASMD (Edit/ASseMble/Debug) which was itself an outgrowth of the Atari Assembler/Editor cartridge. Users of either of these latter two products will find that MAC/65 has a very familiar "feel". Those who have never experienced previous OSS products in this line should nevertheless find MAC/65 to be an easy-to-use, powerful, and adaptable programming environment. While speed was not necessarily the primary goal in the production of this product, we nevertheless feel that the user will be hard pressed to find a faster assembler system in any home computer market. MAC/65 is an excellent match for the size and features of the machines it is intended for.

MAC/65 was conceived by and completely executed by Stephen D. Lawrow. The current version of MAC/65 is only the latest in a series of increasingly more complex and faster assemblers written by Mr. Lawrow following the lead and style of EASMD. As a measure of our confidence in this assembler, it is entrusted with assembling itself, probably a more difficult task than that to which most users will put it.

TRADEMARKS

TABLE OF CONTENTS

INTRODUCTION

This manual assumes the user is familiar with assembly language. It is not intended to teach assembly language. This manual is a reference for commands, statements, functions, and syntax conventions of MAC65 It is also assumed that the user is familiar with the screen editor of the Atari computer. Consult Atari's Reference Manuals if you are not familiar with the screen editor.

If you need a tutorial level manual, we would recommend that you ask your local dealer or bookstore for suggestions.

Although we are hesitant to suggest ANY of the books currently available (because they do not address Atari Computers properly), two books that have worked well for many of our customers are "Machine Language for Beginners" by Richard Mansfield from COMPUTE! books and "Programming the 65Ø2" by Rodney Zaks.

This manual is divided into two major sections. The first two chapters cover the Editor commands and syntax, source line entry, and executing source program assembly. The next three chapters then cover instruction format, assembler directives, functions and expressions, Macros, and conditional assembly.

Note that DDT--the Dunion Debugging Tool--is described in a separate manual section, which follows this MAC/65 manual.

MAC65 is a fast and powerful machine language development tool. Programs larger than memory can be assembled. MAC65 also contains directives specifically designed for screen format development. With MAC65's line entry syntax feature, less time is spent re-assembling programs due to assembly syntax errors, allowing more time for actual program development.

START UP

Simply turn off the power to your computer and insert your MAC/65 cartridge (in the left cartridge slot if using an Atari 800 Computer).

If you are using a disk drive, insert an appropriate DOS boot disk (e.g., DOS XL or Atari DOS) into drive 1 and be sure the drive's power is on.

Turn on your computer. If you have a drive with a proper diskette inserted, DOS will boot. Depending upon the version and kind of DOS you have, you may find that you need to give a command to DOS in order to enter the MAC/65 cartridge. If so, enter the command.

You should be presented with MAC/65's name and copyright lines and an "EDIT" prompt. If not consult your hardware and/or DOS manuals and try again.

You are now ready to begin using MAC/65.

WARM START

The user can exit to DOS XL by entering the MAC/65 command DOS (followed by [RETURN], of course). To return to MAC/65, the user can use the DOS XL command CAR [RETURN] (or menu command 'T').

Unless you have used certain extrinsic commands, DOS XL will return to MAC/65 via a "warm start" (i.e., without clearing out any source lines in memory). Consult your DOS XL manual for details.

Generally, when using Atari DOS, MAC/65 works much like any other cartridge. The MAC/65 "DOS" command will exit to Atari DOS, and the Atari DOS "B" command will return to MAC/65. If you use a MEM.SAV file, your MAC/65 program should stay intact. See your Atari DOS manual for details.

SYNTAX

The following conventions are used in the syntax descriptions in this manual:

1. Capital letters designate commands, instructions, functions, etc., which must be entered exactly as shown (e.g., ENTER, .INCLUDE, .NOT). (But see NOTE below.)

2. Lower case letters specify items which may be used. The various types are as follows:

lno	- Line number between 0-65535, inclusive.
hxnum	- A hex number. It can be address or data. Hex numbers are treated as unsigned integers.
dcnum	- A positive number. Decimal numbers are rounded to the nearest two byte unsigned integer; 3.5 is rounded to 4 and 100.1 to 100.
exp	- An assembler expression.
string	- A string of ASCII characters enclosed by double quotes (eg. "THIS IS A STRING").
strvar	- A string representation. Can be a string, as above, or a string variable within a Macro call (eg. %$1).
filespec OR file	- A string of ASCII characters that refers to a particular device. See device reference manual for more specific explanation.

3. Items in square brackets denote an optional part of syntax (eg. [,lno]). When an optional item is followed by (...) the item(s) may be repeated as many times as needed.

```
Example: .WORD exp [,exp ...]
```

4. Items in parentheses indicate that any one of the items may be used , eg. (,Q) (,A).

NOTE: MAC65 in EDIT mode is NOT case sensitive. Inverse video characters are uninverted. Lower case letters are converted to upper case. EXCEPTIONS: characters between double quotes, following a single quote, or in the comment field of a MAC65 source line will remain unchanged.Text entered in TEXT mode, though, will not be changed.

+

+

---this page intentionally left blank--

+

+

CHAPTER 1: THE EDITOR

The Editor allows the user to enter and edit MAC/65 source code or ordinary ASCII text files.

To the Editor, there is a real distinction between the two types of files; so much so that there are actually two modes accessible to the user, EDIT mode and TEXTMODE. However, for either mode, source code/text must begin with a line number between Ø and 65535 inclusive, followed by one space.

```
Examples: 1Ø LABEL LDA  #$32
          3Ø2Ø This is valid in TEXT MODE
```

The first example would be valid in either EDIT or TEXTMODE, while the second example would only be valid in TEXTMODE.

The user chooses which mode he/she wishes to use for editing by selecting NEW (which chooses the MAC/65 EDIT mode) or TEXT (which allows general text entry). There is more discussion of the impact of these two modes below; but, first, there are several points in common to the two modes.

1.1 GENERAL EDITOR USAGE

The source file is manipulated by Editor commands. Since the Editor recognizes a command by the absence of a line number, a line beginning with a line number is assumed to be a valid source/text line. As such, it is merged with, added to, or inserted into the source/text lines already in memory in accordance with its line number. An entered line which has the same line number as one already in memory will replace the line in memory.

Also, as a special case of the above, a source line can be deleted from memory by entering its line number only. (And also see DEL command for deleting a group of lines.)

Any line that does not start with a line number is assumed to be command line. The Editor will examine the line to determine what function is to be performed. If the line is a valid command, the Editor will execute the command. The Editor will prompt the user each time a command has been executed or terminated by printing:

```
EDIT    for syntax (MAC/65 source) mode
TEXTMODE for text mode
```

The cursor will appear on the following line. Since some commands may take a while to execute, the prompt signals the user that more input is allowed. The user can terminate a command before completion by hitting the break key (escape key on Apple II).

And one last point: If the line is neither a source line or a valid command. The Editor will print:

```
WHAT?
```

1.2 TEXT MODE

The Editor supports a text mode. The text mode is entered with the command TEXT. This mode will NOT syntax check lines entered, allowing the user to enter and edit non-assembly language files. All Editor commands funtion in text mode.

Remember, though, that all text lines must begin with a line number; and, even in TEXTMODE, the space following the line number is necessary.

1.3 EDIT MODE

MAC/65 is nearly unique among assembler/editor systems in that it allows the assembly language user to enter source code and have it IMMEDIATELY checked for syntax validity. Of course, since assembly language syntax is fairly flexible (especially when macros are allowable, as they are with MAC/65), syntax checking will by no means catch all errors in user source code. For example, the existence of and validity of labels and/or zero page locations is not and can not be checked until assembly time. However, we still feel that this syntax checking will be a boon to the beginner and experienced programmer alike.

Again, remember that source lines must begin with a line number which must, in turn, be followed by one space. Then, the second space after the line number is the label column. The label must start in this column. The third space after the line number is the instruction column. Instructions may either start in at least the third column after the line number or at least one space after the label. The operand may begin anywhere after the instruction, and comments may begin anywhere after the operand or instruction. Refer to Assembler Section for specific instruction syntax.

As noted, the Editor syntax checks each source line at entry. If the syntax of a line is in error, the Editor will list the line with a cursor turned on (i.e., by using an inverse or blinking character) at the point of error.

The source lines are tokenized and stored in memory, starting at an address in low memory and building towards high memory. The resultant tokenized file is 60% to 80% smaller than its ASCII counterpart, thus allowing larger programs to be entered and edited in memory.

SPECIAL NOTE: If, upon entry, a source line contains a syntax error and is so flagged by the Editor, the line is entered into Editor memory anyway. This feature allows raw ASCII text files (possibly from other assemblers and possibly containing one or several syntax errors as far as MAC/65 is concerned) to be ENTERed into the Editor without losing any lines. The user can note the lines with errors and then edit them later.

+

+

+

---this page intentionally left blank---

+

+

+

CHAPTER 2: EDITOR COMMANDS

This chapter lists all the valid Editor-level commands, in alphabetical order, along with a short description of the purpose and function of each.

Again, remember that when the "TEXTMODE" or "EDIT" prompt is present any input line not preceded by a line number is presumed to be an Editor command.

If in the process of executing a command any error is encountered, the Editor will abort execution and return to the user, displaying the error number and descriptive message of the error before re-prompting the user. Refer to Appendix for possible causes of errors.

Section 2.1
---

edit command: ASM

purpose : ASseMble MAC/65 source files

usage: ASM [#file1],[#file2],[#file3],[#file4]

ASM will assemble the specified source file and will produce a listing and object code output; the listing may include a full cross reference of all non-local labels. File1 is the source device, file2 is the list device, file3 is the object device, and file4 is a temporary file used to help generate the cross reference listing.

Any or all of the four filespec's may be omitted, in which case MAC/65 assumes the following default filespec(s) are to be used:

file1 - user source memory.
file2 - screen editor.
file3 - memory (CAUTION: see below)
file4 - none, therefore no cross reference

A filespec (#file1, #file3, etc.) can be omitted by substituting a comma in which case the respective default will be used.

For the listing file ONLY, you may use the special form "#-", to indicate that you do NOT want a listing file at all.

Some Examples:

Example: ASM #D2:SOURCE,#D:LIST,#D2:OBJECT

In this example, the source will come from D2:SOURCE, the assembler will list to D:LIST, and the object code will be written to D2:OBJECT.

Example: ASM #D:SOURCE , , #D:OBJECT

In this example, the source will be read from D:SOURCE and the object will be written to D:OBJECT. The assembly listing will be written to the screen.

Example: ASM , #P: , , #D:TEMP

In this example, the source will be read from memory, the object will be written to memory (but ONLY if the ".OPT OBJ" directive is in the source), and the assembly listing will be written to the printer along with the complete label cross reference. The file TEMP on disk drive 1 will be created and used as a temporary file for the cross reference.

Example: ASM #D:SOURCE , #P:

In this example, the source will be read from D:SOURCE and the assembly listing will be written to the printer. If the ".OPT OBJ" directive has been selected in the source, the object code will be placed in memory.

Example: ASM ,#-

This produces what is probably the fastest possible MAC/65 assembly. Source code is read from memory and no listing is produced (because of the "#-"). If your program does not contain a " .OPT OBJ" line, this becomes what is essentially simply an error checking assembly. (Though even if you ARE producing object code, the assembly speed is extremely fast.)

SPECIAL NOTES

Note: If assembling from a "filespec", the source MUST have been a SAVEd file.

Note: Refer to the .OPT directive for specific information on assembler listing and object output.

Note: The object code file will have the format of compound files created by the DOS XL SAVE command. See the DOS XL manual for a discussion of LOAD and SAVE file formats.

Note: You may use #C: as a device for the listing or object files. You may NOT use #C: for the source or cross reference files (thus implying that you may not get a cross reference unless you have a disk drive). HOWEVER, we do not recommend using the cassette as the object file device, since you may get an excessively long leader tone (which will be difficult to re-BLOAD later). Instead, we suggest using BSAVE (after assembling directly to memory) whenever practicable.

Section 2.2
---

edit command: BLOAD

purpose: allows user to LOAD Binary (memory image) files from disk into memory

usage: BLOAD #filespec

The BLOAD command will load a previously BSAVEd binary file, an assembled object file, or a binary file created with OS/A+ SAVe command.

Example: BLOAD #D:OBJECT

This example will load the binary file "OBJECT" to memory at the address where it was previously saved from or assembler for.

Example: BLOAD #C:

This example will load a binary file from cassette.

CAUTION: it is suggested that the user only BLOAD files which were assembled into MAC/65's free area (as shown by the SIZE command) or which will load into known safe areas of memory.

Section 2.3
---

edit command: BSAVE

purpose: SAVE a Binary image of a portion of memory. Same as OS/A+ SAVE command.

usage: BSAVE #filespec < hxnum1 ,hxnum2

The BSAVE command will save the memory addresses from hxnum1 through hxnum2 to the specified device. The binary file created is compatible with the OS/A+ SAVe command.

Example: BSAVE #D:OBJECT<5000,5100

This example will save the memory addresses from $5000 through $5100 to the file "OBJECT".

Example: BSAVE #C: < 5000,5100

This example saves the same memory to cassette.

Section 2.4
---

edit command: BYE

purpose: exit to system monitor level

usage: BYE

BYE will send you to the Atari Memo Pad or your computer's built in diagnostics, depending on which model of computer you have.

Section 2.5
---

edit command: DDT

purpose: enter the DDT debug package which is part of the MAC/65 cartridge.

usage: DDT

Once you have entered this command, DDT is entered and as has control of the system.

However, DDT saves enough of MAC/65's vital memory that, if you follow certain simple rules, you may return to MAC/65 from DDT with your source program still intact.

The DDT manual gives more information on this subject, but as a general guide you must avoid locations $80 through $AF (in zero page) and the memory locations located within the bounds displayed by the SIZE command.

See the DDT manual (which is bound with but after this MAC/65 manual) for many, many more details.

Section 2.6
---

edit command: DEL

purpose: DELetes a line or group of lines from the source/text in memory.

usage: DEL lno1 [,lno2]

DEL deletes source lines from memory. If only one lno is entered, only the line will be deleted. If two lnos are entered, all lines between and including lno1 and lno2 will be deleted.

Note: lno1 must be present in memory for DEL to execute.

Examples:

```
DEL 100          deletes only line 100
DEL 200,1300     deletes lines 200 thru
                         1300, inclusive
```

Section 2.7
---

edit command: DOS [or, equivalently, CP]

purpose: exit from MAC/65 to DOS.

usage:
```
DOS
 or
CP
```

Either DOS or CP returns you to DOS. If you booted an Atari DOS disk, you will be returned to the Atari DOS menu. If you booted DOS XL, you will be returned to either the DOS XL menu or CP (Command Processor), depending upon which was active when you entered MAC/65.

See also the Introduction to this manual for more information on Cold Start and Warm Start as it applies to MAC/65 and the DOS command.

Section 2.8
---

edit command: ENTER

purpose: allow entry of ASCII (or ATASCII) text files into MAC/65 editor memory

usage: ENTER #filespec [(,M) (,A)]

ENTER will cause the Editor to get ASCII text from the specified device. ENTER will clear the text area before entering from the filespec. That is any user program is memory at the time the ENTER command is given will be erased.

The parameter "M" (MERGE) will cause MAC/65 to NOT clear the text area before entering from the file, text entered will be merged with the text in memory. If a line is entered which has the same line number of a line in memory, the line from the device will overwrite the line in memory.

The parameter "A" allows the user to enter un-numbered text from the specified device. The Editor will number the incoming text starting at line 10, in increments of 10.

CAUTION: The "A" option will always clear the text area before entering from the filespec. You may NOT use "M" in conjunction with the "A" option.

Section 2.9
---

edit command: FIND

purpose: to FIND a string of characters somewhere in MAC/65's editor buffer.

usage: FIND /string/ [lno1 [,lno2]] [,A]

The FIND command will search all lines in memory or the specified line(s) (lno1 through lno2) for the "string" given between the matching delimiter. The delimiter may be any character except a space. If a match is found, the line containing the match will be listed to the screen.

Note: do NOT enclose a string in double quotes.

Example: FIND/LDX/

This example will search for the first occurance of "LDX".

Example: FIND\Label\25,80

This example will search for the first occurance of "Label" in lines 25 through 80.

If the option "A" is specified, all matches within the specified line range will be listed to the screen. Remember, if no line numbers are given, the range is the entire program.

Section 2.10
---

edit command: LIST

purpose: to LIST the contents of all or part of MAC/65's editor buffer in ASCII (ATASCII) form to a disk or device.

usage: LIST [#filespec,] [lno1 [,lno2]]

LIST lists the source file to the screen, or device when "#filespec" is specified. If no lnos are specified, listing will begin at the first line in memory and end with the last line in memory.

If only lno1 is specified, that line will be listed if it is in memory. If lno1 and lno2 are specified, all lines between and including lno1 and lno2 will be listed. When lno1 and lno2 are specified, neither one has to be in memory as LIST will search for the first line in memory greater than or equal to lno1, and will stop listing when the line in memory is greater than lno2.

EXAMPLE: LIST #P:
will list the current contents of the editor memory to the P: (printer) device.

EXAMPLE: LIST #D2:TEMP, 1030, 1800
lists only those lines lying in the line number range from 1030 to 1800, inclusive, to the disk file named "TEMP" on disk drive 2.

NOTE: The second example points out a method of moving or duplicating large portions of text or source via the use of temporary disk files. By suitably RENumbering the in-memory text before and after the LIST, and by then using ENTER with the Merge option, quite complex movements are possible.

Section 2.11
---

edit command: LOAD

purpose: to reLOAD a previously SAVEd MAC/65 token file from disk to editor memory.

usage: LOAD #filespec [,A]

LOAD will reload a previously SAVEd tokenized file into memory. LOAD will clear the user memory before loading from the specified device unless the ",A" parameter is appended.

The parameter "A" (for APPEND) causes the Editor to NOT clear the text area before loading from the file. Instead, the load file will be appended with the current file in memory.

Note: The Append option will NOT renumber the file after loading. It is possible to have DUPLICATE LINE NUMBERS. Use the REN command if there are duplicate line numbers.

Section 2.12
---

edit command: LOMEM

purpose: change the lower bound of editor memory usable by MAC/65.

usage: LOMEM hxnum

LOMEM allows the user to select the address where the source program begins.

CAUTION! Executing LOMEM clears out any source currently in memory; as if the user had typed "NEW".

Section 2.13
---

edit command: NEW

purpose: clears out all editor memory, sets syntax checking mode.

usage: NEW

NEW will clear all user source code from memory and reset the Editor to syntax mode. The "EDIT" prompt appears, reminding the user that syntax checking is now active. If the user needs to defeat the syntax checking, he/she must use the TEXT command.

Section 2.14
---

edit command: NUM

purpose: initiates automatic line NUMbering mode

usage: NUM [dcnum1 [,dcnum2]]

NUM will cause the Editor to auto-number the incoming text from the Screen Editor (E:). A space is automatically printed after the line number. If no dcnums are specified, NUM will start at the last line number plus 10. NUM dcnum1 will start at the last line number plus "dcnum1" in increments of "dcnum1". NUM dcnum1, dcnum2 will start at "dcnum1" in increments of "dcnum2".

EXAMPLE: NUM 1000,20
will cause the Editor to prompt the user with the number "1000" followed by a space. When the user has entered a line, the next prompt will be "1020", etc.

The NUM mode will terminate if the line number which would be next in sequence is present in memory.

You may terminate NUM mode by pressing the BREAK key or by typing a CONTROL-3. Optionally, you may press CONTROL-C followed by a [RETURN].

Section 2.15
---

edit command: PRINT

purpose: to PRINT all or part of the Editor text or source to a disk file or a device.

usage: PRINT [#filespec,] [lno1 [,lno2]]

Print is exactly like LIST except that the line numbers are not listed. If a file is PRINTed to a disk, it may be reENTERed into the MAC/65 memory using the ENTER command with the Append line number option.

Section 2.16
---

edit command: REN

purpose: RENumber all lines in Editor memory.

usage: REN [dcnum1 [,dcnum2]]

REN renumbers the source lines in memory. If no dcnums are specified, REN will renumber the program starting at line 10 in increments of 10. REN dcnum1 will renumber the lines starting at line 10 in increments of dcnum1. REN dcnum1, dcnum2 will renumber starting at dcnum1 in increments of dcnum2.

Section 2.17
---

edit command: REP

purpose: REPlaces occurrence(s) of a given string with another given string.

usage:
REP /old string/new string/ [lnol [,lno2]] [(,A)(,Q)]

The REP command will search the specified lines (all or lnol through lno2) for the "old string".

The "A" option will cause all occurrences of "old string" to be replaced with "new string". The "Q" option will list the line containing the match and prompt the user for the change (Y followed by RETURN for change, RETURN for skip this occurrance.) If neither "A" or "Q" is specified, only the first occurrence of "old string" will be replaced with "new string". Each time a change is made, the line is listed.

Example: REP/LDY/LDA/200,250,Q

This example will search for the string "LDY" between the lines 200 and 250, inclusive, and prompt the user at each occurrence to change or skip.

Note: Hitting BREAK (ESCape on Apple II) will terminate the REP mode and return to the Editor.

Note: If a change causes a syntax error in the line, the REP mode will be terminated and control will return to the Editor. Of course, if TEXTMODE is selected, there can be no syntax errors.

Section 2.18
---

edit command: SAVE

purpose: SAVEs the internal (tokenized) form of the user's in-memory text/source to a disk file.

usage: SAVE #filespec

SAVE will save the tokenized user source file to the specified device. The format of a tokenized file is as follows:

File Header
Two byte number (LSB,MSB) specifies the size of the file in bytes.

For each line in the file:
Two byte line number (LSB,MSB)
followed by
One byte length of line (actually offset to next line)
followed by
The tokenized line

Section 2.19
---

edit command: SIZE

purpose: determines and displays the SIZE of various portions of memory used by the MAC/65 Editor.

usage: SIZE

SIZE will print the user LOMEM address, the highest used memory address, and the highest usable memory address, in that order, using hexadecimal notation for the addresses.

These memory addresses are especially helpful in determining what areas of memory to avoid when assembling programs directly to memory. Remember, though, that MAC/65 needs a certain amount of room above the middle address shown for the symbol table (when an assembly is made). See also the DDT manual for hints on memory usage.

Section 2.20
---
edit command: TEXT

purpose: allow entry of arbitrary ASCII (ATASCII) text without syntax checking.

usage: TEXT

TEXT will clear all user source code from memory and put the Editor in the text mode. After this command is used, the Editor will prompt the user for new commands and text with the word "TEXTMODE" (instead of "EDIT"), indicating that no syntax checking is taking place.

TEXTMODE may be terminated by the NEW command. CAUTION: there is no way to go back and forth between syntax (EDIT) mode and TEXTMODE without clearing the Editor's memory each time.

Section 2.21
---
edit command: ?

purpose: makes hexadecimal/decimal conversions

usage: ? ($hxnum) (dcnum)

? is the resident hex/decimal decimal/hex converter. Numbers in the range 0 - 65535 decimal (0000 to FFFF hex) may be converted.

Example: ? $1200 will print =4608
? 8190 will print =$1FFE

+

+

---this page intentionally left blank--

+

+

CHAPTER 3: THE MACRO ASSEMBLER

The Assembler is entered from MAC/65 with the command ASM. For ASM command syntax, refer to section 2.1 (in the Editor commands). Assembly may be terminated by hitting the BREAK key. MAC/65 properly closes files and "cleans up" before terminating the assembly.

3.1 ASSEMBLER INPUT

The Assembler will get a line at a time from the specified device or from memory. If assembling from a device, the file must have been previously SAVEd by the Editor. All discussions of source lines and syntax will be at the Editor line entry level. The tokenized (SAVEd) form is discussed in general terms under the SAVE command, section 2.19.

Source lines are in the form:

line number + mandatory space + source statement

The source statement may be in one of the following forms:

[label] [(6502 instruction) (directive)] [comment]

The following examples are valid source lines:

```
100 LABEL
120 ;Comment line
140   LDA  #5   and then any comment at all
150   DEY
160   ASL A     double number in accumulator
170 GETNUM LDA (ADDRESS),Y
180  .PAGE  "directives are legal, too"
```

In general, the format is as specified in the MOS Technology 6502 Programing Manual. We recommend that the user unfamiliar with 6502 assembly language programming should purchase:

"Machine Language for Beginners" by R. Mansfield
or
"Programing the 6502" by Rodney Zaks
or
any other book which seems compatible with the users current knowledge of assembly language.

SPECIAL NOTE: The assembler of MAC/65 understands only upper case labels, op codes, etc. HOWEVER, the editor (see expecially section 1.3) will convert all lower case to upper case (except in comments and quoted strings), so the user may feel free to type and edit in whichever case he/she feels most comfortable with.

3.2 INSTRUCTION FORMAT

A) Instruction mnemonics are as described in the MOS Technology Programing Manual.

B) Immediate operands begin with "#".

C) "(operand,X)" and "(operand),Y" designate indexed indirect and indirect indexed addressing, respectively.

D) "operand,X" and "operand,Y" designate indexed addressing.

E) Zero page operands cannot be forward referenced. Attempting to do so will usually result in a "PHASE ERROR" message.

F) Forward equates are evaluated within the limits of a two pass assembler.

G) "*" designates the current location counter.

H) Comment lines may begin with ";" or "*".

I) A semicolon (";") anywhere in a line indicates the beginning of the comment field for that line.

J) Hex constants begin with "$".

K) The "A" operand is reserved for accumulator addressing.

L) The addressing formats available are extended to allow the new addressing modes available with the NCR 65CØ2 microprocessor. See Chapter 7 for the descriptions of 65CØ2 instructions not included in the standard 65Ø2 set. The extensions include:

1. "(operand)", indicating indirect addressing, is now legal with ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA. The operand must be in zero page.

2. "(operand,X)" is now legal when used with JMP. The operand here may be any absolute address.

3. The BIT instruction is allowed the addressing mode "operand,x". The operand may be either a zero page or absolute address.

4. The mnemonics BRA, DEA, INA, PHX, PHY, PLX, PLY, STZ, TRB, and TSB are now recognized.

3.3 LABELS

Labels must begin with an Alpha character, "@", or "?". The remaining characters may be as the first or may be "Ø" to "9" or ".". The characters must be uppercase (but remember that the editor always converts lowercase for you) and cannot be broken by a space. The maximum number of characters in a label is 127, and ALL are significant.

Labels beginning with a question mark ("?") are assumed to be "LOCAL" labels. Such labels are "visible" only to code encountered within the current local region. Local regions are delimited by successive occurrences of the .LOCAL directive, with the first region assumed to start at the beginning of the assembly source, whether or not a .LOCAL is coded there or not. There are a maximum of 62 local regions in any one assembly. Of course, if a .LOCAL is not encountered anywhere in the assembly, then all labels are accessible at all times. In any case, labels beginning with a question mark will NOT be listed in the symbol table.

The following are examples of valid labels:

```
TEST1  @.INC  LOCATION  LOC22A  WHAT?
ADDRESS1.1  EXP..  SINE45TAB.
```

3.4 OPERANDS

An operand can be a label, a Macro parameter, a numeric constant, the current program counter (*), "A" for accumulator addressing, an expression, or an ASCII character preceded by a single quote (e.g., '?). The following are examples of the various types of operands:

```
1Ø      LDA     #VALUE          ; label
15      ROR     A               ; accumulator addressing
2Ø      .BYTE   123,$45         ; numeric constants
25      .IF     %Ø              ; Macro parameter
3Ø      CMP     #'A             ; ASCII character
35  THISLOC = *                 ; current PC
4Ø      .WORD   PMBASE+[PLNO+4]*256   ; expression
```

3.5 OPERATORS

The following are the operators currently supported by MAC/65:

[ ]	pseudo parentheses
+	addition
-	subtraction
/	division
\	modulo (remainder after integer division)
*	multiplication
&	binary AND
!	binary OR
^	binary EOR
=	equality, logical
>	greater than, logical
<	less than, logical
<>	inequality, logical
>=	greater or equal, logical
<=	less or equal, logical
.OR	logical OR
.AND	logical AND
-	unary minus
.NOT	unary logical. Returns true (1) if expression is zero. Returns false (Ø) if expression is non-zero.
.DEF	unary logical label definition. Returns true if label is defined.
.REF	unary logical label reference. Returns true if label has been referenced.
>	unary. Returns the high byte of the expression.
<	unary. Returns the low byte of the expression.

Logical operators will always return either TRUE (1) or FALSE (Ø). However, any non-zero value is considered true when making a conditional test. Also, undefined labels are given a value of zero (False).

Some of these operators perhaps need some explanation as to their usage and purpose. The operators are thus described in groups in the following subsections.

3.5.1 Operators: + - * / \

These are the familiar arithmetic operators, though "\" may be new to you, even if the modulus operation is not. Remember, though, that they perform 16-bit signed arithmetic and ignore any overflows. Thus, for example, the value of $FF00+4096 is $0F00, and no error is generated.

```
COMMENT:  "op1 \ op2" is exactly equivalent to
          "op1 - [ op2 * [ op1 / op2 ] ]"
          and is the remainder after integer division
          is performed.  Example: 11\4 is 3.
```

3.5.2 Operators: & ! ^

These are the binary or "bitwise" operators. They operate on values as 16 bit words, performing bit-by-bit ANDs, ORs, or EXCLUSIVE ORs. They are 16 bit equivalents of the 6502 opcodes AND, ORA, and EOR.

```
EXAMPLES:        $FF00 & $00FF  is $0000
                 $03 ! $0A      is $000B
                 $003F ^ $011F  is $0120
```

3.5.3 Operators: = > < <> >= <=

These are the familiar comparison operators. They perform 16 bit unsigned compares on pairs of operands and return a TRUE (1) or FALSE (0) value.

```
EXAMPLES:        3 < 5    returns 1
                 5 < 5    returns 0
                 5 <= 5   returns 1
```

CAUTION: Remember, these operators always work on PAIRS of operands. The operators ">" and "<" have quite different meanings when used as unary operators.

3.5.4 Operators: .OR .AND .NOT

These operators also perform logical operations and should not be confused with their bitwise companions. Remember, these operators always return only TRUE or FALSE.

```
EXAMPLES:        3 .OR 0     returns 1
                 3 .AND 2    returns 1
                 6 .AND 0    returns 0
                 .NOT 7      returns 0
```

3.5.5 Operator: - (unary)

The minus sign may be used as a unary operator. Its effect is the same as if a minus sign had been used in a binary operation where the first operator is zero.

```
EXAMPLE:        -2  is  $FFFE (same as Ø-2)
```

3.5.6 Operators: < > (unary)

These UNARY operators are extremely useful when it is desired to extract just the high order or low order byte of an expression or label. Probably their most common use will be that of supplying the high and low order bytes of an address to be used in a "LDA #" or similar immediate instruction.

```
EXAMPLE:        FLEEP = $3456
                 LDA #<FLEEP (same as LDA #$56)
                 LDA #>FLEEP (same as LDA #$34)
```

3.5.7 Operator: .DEF

This unary operator tests whether the following label has been defined yet, returning TRUE or FALSE as appropriate.

CAUTION: Defining a label AFTER the use a .DEF which references it can be dangerous, particularly if the .DEF is used in a .IF directive.

```
EXAMPLE:          .IF .DEF ZILK
                  .BYTE "generate some bytes"
                  .ENDIF
                ZILK = $3ØØØ
```

In this example, the .BYTE string will NOT be generated in the first pass but WILL be generated in the second pass. Thus, any following code will almost undoubtedly generate a PHASE ERROR.

3.6 ASSEMBLER EXPRESSIONS

An expression is any valid combination of operands and operators which the assembler will evaluate to a 16-bit unsigned number with any overflow ignored. Expressions can be arithmetric or logical. The following are examples of valid expressions:

```
10    .WORD   TABLEBASE+LINE*COLUNM
55    .IF  .DEF INTEGER .AND [ VER=1 .OR VER >=3 ]
200   .BYTE   >EXPLOT-1,  >EXDRAW-1,  >EXFILL-1
300   LDA     # < [ < ADDRESS^-1 ] + 1
305   CMP     # -1
400   CPX     # 'A
440   INC     $1+1
```

3.7 OPERATOR PRECEDENCE

The following are the precedence levels (high to low) used in evaluating assembler expressions:

```
[ ] (pseudo parenthesis)
> (high byte),  < (low byte),  .DEF,  .REF,  - (unary)
.NOT
*,  /, \
+,  -
&,  !,  ^
=,  >,  <,  <=,  >=,  <>          (comparison operators)
.AND
.OR
```

Operators grouped on the same line have equal precedence and will be executed in left-to-right order unless higher precedence operator(s) intervene.

Generally, the operator precedences are what you would expect on a mathematical basis. Care must be taken, however, with the '<' and '>' unary operators.

For Example:

```
TABLE = $45FE
        LDA # > TABLE + 3 ; A receives $48
        LDA # > [TABLE+3] ; A receives $46
```

The only operator which can legally combined with .REF is .NOT, as in .IF .NOT .REF LABEL.

Note that the illegal line above could be simulated thus:

EXAMPLE:

```
DOIT .= Ø
  .IF .REF ZAM
DOIT .= 1
  .ENDIF
  .IF .REF BLOOP
DOIT .= 1
  .ENDIF
  .IF DOIT
  ...
```

3.5.9 Operator: [ ]

MAC/65 supports the use of the square brackets as "pseudo parentheses". Ordinary round parentheses may NOT be used for grouping expressions, etc., as they must retain their special meanings with regards to the various addressing modes. In general, the square brackets may be used anywhere in a MAC/65 expression to clarify or change the order of evaluation of the expression.

EXAMPLES:

```
        LDA GEORGE+5*3      ; This is legal, but
                              it multiplies 3*5
                              and adds the 15 to
                              GEORGE...probably
                              not what you wanted.
        LDA (GEORGE+5)*3    ; Syntax Error!!!
        LDA [GEORGE+5]*3    ; OK...the addition
                              is performed before
                              the multiplication
        LDA ( [GEORGE+5]*3 ),Y  ; See the need
                              for both kinds of
                              "parentheses"?
```

REMEMBER: Operators in MAC/65 expressions follow precedence rules. The square brackets may be used to override these rules.

CHAPTER 4: DIRECTIVES

As noted in Section 3.1, the instruction field of an assembled line may contain an assembler directive (instead of a valid 6502 instruction). This chapter will list and describe, in roughly alphabetical order, all the directives legal under MAC/65 (excepting directives specific to macros, which will be discussed separately in Chapter 5).

Directives may be classified into three types: (1) those which produce object code for use by the assembled program (e.g., .BYTE, .WORD, etc.); (2) those which direct the assembler to perform some task, such as changing where in memory the object code should go or giving a value to a label (e.g., *=, =, etc.); and (3) those which are provided for the convenience of the programmer, giving him/her control over listing format, location of source, etc. (e.g., .TITLE, .OPT, .INCLUDE).

Obviously, we could in theory do without the type 3 directives; but, as you read the descriptions that follow, you will soon discover that in practice these directives are most useful in helping your 6502 assembly language production. Incidentally, all the macro-specific directives could presumably be classified as type 3.

Three of the directives which follow (.PAGE, .TITLE, and .ERROR) allow the user to specify a string (enclosed in quotes) which will be printed out. For these three directives, the user is limited to a maximum string length of 70 characters. Strings longer than 70 characters will be truncated.

3.8 NUMERIC CONSTANTS

MAC/65 accepts three types of numeric constants: decimal, hexadecimal, and characters.

A decimal constant is simply a decimal number in the range 0 through 65535; an attempt to use a decimal number beyond these bounds may or may not work and will certainly produce unexpected and undesired results.

```
EXAMPLES:       1   234   65200  32767
(as used:)      .BYTE 2,4,8,16,32,64
                LDA #1
```

A hexadecimal constant consists of a dollar sign followed by one to four legal hexadecimal digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F). Again, usage of more than four digits may produce unwanted results.

```
EXAMPLES:       $1  $EA  $FF00  $7FFF
(as used:)      .WORD $100,$200,$400,$800,$1000
                AND #$7F
```

A character constant is an apostrophe followed by any printable or displayable character. The value of a character constant is the ASCII (or ATASCII) value of the character following the apostrophe.

```
EXAMPLES:       'A   '*   '"   '=
(as used:)      CMP  #'=
                CMP  #'Z+1 ; same as #$5B
                CMP  #'J+3 ; same as #'M
```

3.9 STRINGS

Strings are of two types. String literals (example: "This is a string literal"), and string variables for Macros (example: %$5).

```
Example:  10    .BYTE  "A STRING OF CHARACTERS"
                         or
Example:  20    .SBYTE %$1
```

NOTE that there are really only six places where a string is legal in MAC/65: as a parameter to a called macro or as the operand to .BYTE, .CBYTE, .SBYTE, .TITLE, or .PAGE.

Section 4.2
---

directive: = and .EQU

purpose: assigns a value to a label

usage: label = expression
label .EQU expression

The "=" directive will equate "label" with the value of the expression. A "label" can be equated via "=" only once within a program.

```
Example:  10  PLAYER0  =  PMBASE + $200
          20  PLAYER1 .EQU PMBASE + $280
```

Note: If a "label" is equated more than once, "label" will contain the value of the most recent equate. This process will, however, result in an assembly error.

Section 4.3
---

directive: .=

purpose: assign a possibly transitory value to a label

usage: label .= expression

The .= directive will SET "label" with the value of the expression. Using this directive, a "label" may be set to one or more values as many times as needed in the same program.

```
EXAMPLE:
10 LBL  .=   5
20      LDA  #LBL       ; same as LDA #5
30 LBL  .=   3+'A
40      LDA  #LBL       ; same as LDA #68
```

CAUTION: A label which has been equated (via the "=" directive) or assigned a value through usage as an instruction label may not then be set to another value by ".=".

Section 4.1
---

directive: *= and .ORG

purpose: change current origin of the assembler's location counter

usage: [label] *= expression
[label] .ORG expression

The *= (or, equivalently, .ORG) directive will assign the value of the expression to the location counter. The expression cannot be foward referenced. (*= must be written with no intervening spaces.)

```
Example:  50    *= $1234     ; sets the location
                               counter to $1234
          135   .ORG $1234   ; ditto
```

Another common usage of *= is to reserve space for data to be filled in or used at run time. Since the single character "*" may be treated as a label referencing the current location counter value, the form "*= *+exp" is thus the most common way to reserve "exp" bytes for later use.

```
Example:  70 LOC *= *+1 ; assigns the current
                          value of the location
                          counter to LOC and
                          then advances the
                          counter by one.
          70 LOC .ORG *+1 ; ditto
```

(Thus LOC may be thought of as a one byte reserved memory cell.)

CAUTION: Because any label associated with this directive is assigned the value of the location counter BEFORE the directive is executed, it is NOT advisable to give a label to "*=" or ".ORG" unless, indeed, it is being used as in the second example (i.e., as a memory reserver).

NOTE: Some assemblers treat the label on an "ORG" or ".ORG" directive differently. That is, they assign the Label to the location counter AFTER it has been changed by the directive. Use caution when converting from and to such assemblers; pay special attention to label usage. When in doubt, move the label to the next preceding or next following line, as appropriate.

SPECIAL NOTE: Although the form "label *= *+exp" is standard 6502 usage, you may find MAC/65's ".DS" directive (section 4.7) easier to read and understand.

3.5.8 Operator: .REF

This unary operator tests whether the following label has been referenced by any instruction or directive in the assembly yet; and, in conjunction with the .IF directive, produces the effect of returning a TRUE or FALSE value.

Obviously, the same cautions about .DEF being used before the label definition apply to .REF also, but here we can obtain some advantage from the situation.

EXAMPLE:

```
        .IF .REF PRINTMSG
PRINTMSG
  ... (code to implement
       the PRINTMSG
       routine)
  .ENDIF
```

In this example, the code implementing PRINTMSG will ONLY be assembled if something preceding this point in the assembly has referred to the label PRINTMSG! This is a very powerful way to build an assembly language library and assemble only the needed routines. Of course, this implies that the library must be .INCLUDEd as the last part of the assembly, but this seems like a not too onerous restriction. In fact, OSS has used this technique in writing the libraries for the C/65 compiler.

CAUTION: note that in the description above it was implied that .REF only worked properly with a .IF directive. Not only is this restriction imposed, but attempts to use .REF in any other way can produce bizarre results. ALSO, .REF cannot effectively be used in combination with any other operators. Thus, for example,

```
.IF .REF ZAM  .OR  .REF BLOOP   is ILLEGAL!
```

Section 4.4
---

directive: .BYTE [and .SBYTE]

purpose: specifies the contents of individual bytes in the output object

usage:
[label] .BYTE [+exp,] (exp)(strvar) [,(exp)(strvar) ...]
[label] .SBYTE [+exp,] (exp)(strvar) [,(exp)(strvar) ...]

The .BYTE and .SBYTE directives allow the user to generate individual bytes of memory image in the output object. Expressions must evaluate to an 8-bit arithmetic result. A strvar will generate as many bytes as the length of the string. .BYTE simply assembles the bytes as entered, while .SBYTE will convert the bytes to Atari screen codes.

```
Example:  100    .BYTE "ABC" , 3 , -1
```

This example will produce the following output bytes:
41 42 43 03 FF.
Note that the negative expression was truncated to a single byte value.

```
Example:  50    .SBYTE "Hello!"
```

On the Atari, this example will produce the following screen codes:
28 65 6C 6C 6F 01.

SPECIAL NOTE: Both .BYTE and .SBYTE allow an additive Modifier. A Modifier is an expression which will be added to all of bytes assembled. The assembler recognizes the Modifer expression by the presence of the "+" character. The Modifier expression will not itself be generated as part of the output.

```
Example:  5    .BYTE +$80 , "ABC" , -1
```

This example will produce the following bytes:
C1 C2 C3 7F.

Example: 100 .BYTE +$80,"DEF",'G+$80

This example will produce: C4 C5 C6 47.

(Note especially the effect of adding $80 via the modifier and also adding it to the particular byte. The result is an unchanged byte, since we have added a total of 256 ($100), which does not change the lower byte of a 16 bit result.)

Example: 55 .SBYTE +$40 , "A12"

This example will produce: 61 51 52

Example: 80 .SBYTE +$C0,'G-$C0,"REEN"

This example will produce: 27 F2 E5 E5 EE

Note: .SBYTE performs its conversions according to a numerical algorithm and does NOT special case any control characters, including BELL, TAB, etc.--these characters ARE converted.

Section 4.5
---

directive: .CBYTE

purpose: same as .BYTE except that the most significant bit of the last byte of a string argument is inverted

usage:
[label] .CBYTE [+exp,] (exp)(strvar) [,(exp)(strvar)...]

The .CBYTE directive may often be used to advantage when building tables of strings, etc., where it is desirable to indicate the end of a string by some method other than, for example, storing a following zero byte. By inverting the sense of the upper bit of that last character of the string, a routine reading the strings from the table could easily do a BMI or BPL as it reads each character.

Example: ERRORS .CBYTE 1,"SYSTEM"

The line shown would produce these object bytes:
01 53 59 53 54 45 CE

(continued on next page)

(.CBYTE, continued)

And a subroutine might access the characters thus:

```
            LDY #1
LOOP        LDA ERRORS,Y
            BMI ENDOFSTRING
            INY
            BNE LOOP
            ...
ENDOFSTRING
            ...
```

Section 4.6

directive: .DBYTE [see also .WORD]

purpose: specifies Dual BYTE values to be placed in the output object.

usage: [label] .DBYTE exp [,exp ...]

Both the .WORD and .DBYTE directives will put the value of each expression into the object code as two bytes. However, while .WORD will assemble the expression(s) in 6502 address order (least significant byte, most significant byte), .DBYTE will assemble the expression(s) in the reverse order (i.e., most significant byte, least significant byte).

.DBYTE has limited usage in a 6502 environment, and it would most probably be used in building tables where its reversed order might be more desirable.

```
EXAMPLE:  .DBYTE  $1234,1,-1
                  produces:  12 34 00 01 FF FF
          .WORD   $1234,1,-1
                  produces:  34 12 01 00 FF FF
```

Section 4.7

directive: .DS

purpose: reserves space for data without initializing the space to any particular value(s).

usage: [label] .DS expression

Using " .DS expression" is exactly equivalent to using " *= *+expression". That is, the label (if it is given) is set equal to the current value of the location counter. Then the value of the expression is added to the location counter.

```
Example:  BUFFERLEN .DS 1 ; reserve a single byte
          BUFFER    .DS 256 ; reserve 256 bytes
```

Section 4.8
---
directive: .ELSE

purpose: SEE description of .IF for purpose and usage.

Section 4.9
---
directive: .END

purpose: terminate an in-memory assembly

usage: [label] .END

The .END directive will terminate the assembly ONLY if the source is being read from memory. Otherwise, .END will have no effect on assembly.

This "no effect" is handy in that you may thus .INCLUDE file(s) without having to edit out any .END statements they might contain. In truth, .END is generally not needed at all with MAC/65.

Section 4.10
---
directive: .ENDIF

purpose: terminate a conditional assembly block

SEE description of .IF for usage and details.

Section 4.11
---
directive: .ERROR

purpose: force an assembler error and message

usage: [label] .ERROR [string]

The .ERROR directive allows the user to generate a pseudo error. The string specified by .ERROR will be sent to the screen as if it were an assembler-generated error. The error will be included in the count of errors given at the end of the assembly.

Example: 100 .ERROR "MISSING PARAMETER!"

Section 4.12
---

directive: .FLOAT

purpose: specifies floating point constant values to be placed in the output object.

usage:
[label] .FLOAT floating-constant [,floating-constant...]

This directive would normally only be used by the programmer wishing to access the built-in floating point routines of the Atari Operating System ROM's.

Each floating point constant following the .FLOAT directive will produce 6 bytes of output object code, in a format consistent with the above-mentioned floating point routines. In particular, the first byte contains the exponent portion of the number, in excess-64 notation representing powers of 100. The upper bit of the exponent byte designates the sign of the mantissa portion. The following 5 bytes are the mantissa, in packed BCD form, normalized on a byte boundary (consistent with the powers-of-100 exponent).

EXAMPLES:

```
.FLOAT 3.14156295,-2.718281828
```

The above example would produce the following bytes in the output object code:

```
40 03 14 15 62 95
C0 27 18 28 18 28
```

NOTE: Only floating point constants, NOT expressions, are legal as operands to .FLOAT. Generally, this is not a problem, since the user may perform any constant arithmetic on a calculator (or in BASIC) before placing the result in his/her MAC/65 program.

Section 4.13
---

directive: .IF

purpose: chooses to perform or not perform some portion of an assembly based on the "truth" of an expression.

usage:
```
.IF     exp
[.ELSE]
.ENDIF
```

usage note: there may be any number of lines of assembly language code or directives between .IF and .ELSE or .ENDIF and similarly between .ELSE and .ENDIF.

The .IF, .ELSE, and .ENDIF directives control conditional assembly.

When a .IF is encountered, the following expression is evaluated. If it is non-zero (TRUE), the source lines following .IF will be assembled, continuing until an .ELSE or .ENDIF is encountered. If an .ELSE is encountered before an .ENDIF, then all the source lines between the .ELSE and the corresponding .ENDIF will not be assembled. If the expression evaluates to zero (false), the source lines following .IF will not be assembled. Assembly will resume when a corresponding .ENDIF or an .ELSE is encountered.

The .IF-.ENDIF and .IF-.ELSE-.ENDIF constructs may be nested to a depth of 14 levels. When nested, the "search" for the "corresponding" .ELSE or .ENDIF skips over complete .IF-.ENDIF constructs if necessary.

Examples:

```
10      .IF  1              ; non-zero, therefore true
20      LDA  # '?           ; these two lines will
30      JSR  CHAROUT        ; be assembled
40      .ENDIF
```

EXAMPLE:

```
10      .IF  Ø                ; expression is false
11      LDX  # >ADDRESS       ; these two lines will
12      LDY  # <ADDRESS       ; not be assembled
13      .IF  1
14      .ERROR "can't get here"
15 ; likewise, this can't be assembled because it
16 ; is "nested" within the .IF Ø structure
17 ;
18      .ELSE
19 ;
20      LDX  # <ADDRESS       ; these lines will
21      LDA  # >ADDRESS       ; be assembled
22      .ENDIF
23      JSR          PRINTSTRING    ; go print the string
```

Note: The assembler resets the conditional stack at the begining of each pass. Missing .ENDIF(s) will NOT be flagged.

Section 4.14
---

directive: .INCLUDE

purpose: allows one assembly language program to request that another program be included and assembled in-line

usage: .INCLUDE #filespec

usage note: this directive should NOT have a label

The .INCLUDE directive causes the assembler to begin reading source lines from the specified "filespec". When the end of "filespec" is reached, the assembler will resume reading source from the previous file (or memory).

CAUTION: The .INCLUDEd file MUST be a properly SAVEd MAC/65 tokenized program. It can NOT be an ASCII file.

Note: A .INCLUDED file cannot itself contain a .INCLUDE directive.

EXAMPLE: .INCLUDE #D:SYSEQU.M65

This example line will include the system equates file supplied by OSS.

Section 4.15
---

directive: .LOCAL

purpose: delimits a local label region

usage: .LOCAL

usage note: this directive should not be associated with a label.

This directive serves to end the previous local region and begin a new local region. It is assumed that the first local region begins at the beginning of the assembly, and the last local region ends at the end of the assembly.

Within each local region, any label beginning with a colon (":") or question mark ("?") is assumed to be a "local label". As such, it is invisible to code, equates, references, etc., outside of its own local region.

This feature is especially handy when using automatic code generators or when several people are working on a single project. In both these cases, the coder may use labels beginning with ":" or "?" and be sure that there will be no duplicate label errors produced.

EXAMPLE:

```
10  *= $4000
11  LDX #3    ; establish a counter
12 ?LOOP
13  LDA FROM,X ; get a byte
14  STA TO,X   ; put a byte
15  DEX        ; more to do?
16  BPL ?LOOP ; goes to label on line 12
17 ;
18  .LOCAL      ; another local region!
19 ;
20 ?LOOP = 6
21 ;
22  LDY #?LOOP  ; same as LDY #6
23  (etc.)
```

FEATURE: Local labels MAY be forward referenced, just like any other label.

NOTE: Local labels do not appear in the symbol table listing. Except see Chapter 9.

Section 4.16
---

directive: .OPT

purpose: selects various assembly control OPTions

usage: .OPT option [, [NO] option ...]
(or)
.OPT NO option [, [NO] option ...]

usage notes: the valid options are as follows:

LIST	ERR	EJECT	OBJ
MLIST	CLIST	NUM	XREF

The .OPT directive allows the user to control certain functions of the assembly. Generally, coding ".OPT option" will invoke a feature or option, while ".OPT NO option" will "turn off" that same feature.

You may use any number of options (or NO options) on a single source line. For example, it is legal to use:

```
.OPT NO LIST, NO XREF, OBJ, ERR
```

The following are the descriptions of the individual options:

LIST controls the entire assembly listing.
NO LIST turns off all listing except error lines.

ERR will determine if errors are returned to the user in the listing and/or the screen.
NO ERR is thus dangerous.

EJECT controls the title and page listing.
NO EJECT only turns off the automatic page generation; it has no effect on .PAGE requests.

OBJ determines if the object code is written to the device/memory.
NO OBJ is useful during trial assemblies.
OBJ is NECESSARY when the object code is to placed in memory.

NUM will auto number the assembly listing instead of using the user line numbers. NUM will begin at 100 and increment by 1.
NUM is generally not useful except for final, "pretty" assemblies.

MLIST controls the listing of Macro expansions. NO MLIST will list only the lines within a Macro expansion which generate object code. MLIST will expand the entire Macro.

Note that NO MLIST is extraordinarly useful in producing readable listings.

CLIST controls the listing of conditional assembly. NO CLIST will not list source lines which are not assembled. CLIST will list all lines within the conditional construct.

XREF allows the user, when a cross reference has been specified in the ASM command line, to control which portions of the source program will be cross referenced during the assembly.

Any lines of source code between a .OPT NO XREF and the next suceeding .OPT XREF will not be cross referenced.

By combining NO XREF and NO LIST, you can list and cross reference even extremely large programs in pieces. Or you might use NO XREF to avoid indexing entries out of an INCLUDEd file. XREF and NO XREF are useless and inoperative (but do not generate errors) if you have not specified a cross reference file name in the ASM command line.

NOTE: Unless specified otherwise by the user, all of the options will assume their default settings. The default settings for .OPT are:

LIST	listing IS produced
ERR	errors are reported
EJECT	pages are numbered and ejected
NO NUM	use programmer's line numbers
MLIST	all macro lines are listed
CLIST	all failed conditionals list
XREF	continous cross reference
NO OBJ	SEE CAUTION !!!!!

CAUTION: The OBJ option is handled in a special way:
IF assembling to memory the object default is NO OBJ.
IF assembling to a device the object option is OBJ.

NOTE: Macro expansions with the NO NUM option will not be listed with line numbers.

Section 4.17

directive: .PAGE

purpose: provides page headings and/or moves to top of next page of listing

usage: .PAGE [string]

usage note: no label should be used with .PAGE

The .PAGE directive allows the user to specify a page heading. The page heading will be printed below the page number and title heading.

.PAGE will eject the next page, and prints the most recent title and page headings.

```
Example:   300    .PAGE  "EXECUTE LABEL SEARCH"
```

Note: The assembler will automatically eject and print the current title and page headings after 61 lines have been listed.

Section 4.18

directive: .SBYTE

purpose: produces "screen" bytes in output object

usage: see .BYTE description, section 4.4

Section 4.19
---

directive: .SET

purpose: controls various assembler functions

usage: .SET dcnum1 , dcnum2

The .SET directive allows the user to change specific variable parameters of the assembler. The dcnum1 specifys the parameter to change, and dcnum2 is the changed value. The following table summarizes the various .SET parameters. Defaults for each parameter are given in parentheses, followed by the allowable range of values.

dcnum1	dcnum2	function
Ø	(4) 1-4	sets the .BYTE and .SBYTE listing format. 1 to 4 bytes can be printed in the object code field of the listing.
1	(Ø) Ø-31	sets the assembly listing left margin. The specified number is the number of spaces which will be printed before the assembled source line.
2	(8Ø) 4Ø-132	set width for listing, adjust for your printer.
3	(12) Ø,12	form feed select. Ø implies no form feed on printer--use multiple line feeds. Any other used as form feed char.
4	(66) 2Ø-255	number of lines per page for listing.
5	(Ø) Ø-255	number of spaces from semicolon in comment field to where remainder of comment is printed.
6	(Ø) Ø-$FFFF	an offset, which is added to the location counter when an object byte is stored or written to disk. You can thus assemble code for one address while storing or loading it another address.

***** SPECIAL NOTE: See Chapter 8 for a complete ***** discussion of the capabilities of .SET 6

Section 4.20

directive: .TAB

purpose: sets listing "tab stops" for readability

usage: .TAB dcnum1 ,dcnum2 ,dcnum3

The .TAB directive allows the user to specify the starting column for the listing of the instruction field, the operand field, and the comment field respectively. The defaults are 8,12,20.

```
Example:  200    .TAB 16,32,50
          ...
          1200   .TAB 8,12,20 ; restores defaults
```

Section 4.21

directive: .TITLE

purpose: specify assembly listing heading

usage: .TITLE string

The .TITLE directive allows the user to specify a assembly title heading. The title string will be printed at the top of every page following the page number.

Section 4.22

directive: .WORD [see also .DBYTE]

purpose: place 16 bit word values in output object

usage: [label] .WORD exp [,exp ...]

The .WORD and .DBYTE directives both put the value of each following expression into the object code as two bytes. But where .WORD will assemble the expression(s) in 6502 address order (least significant byte, most significant byte), .DBYTE will assemble the expression(s) in reverse order (most significant byte, least significant byte).

Generally, for 6502 programs, .WORD is the more useful of the two, and is more compatible with the code produced by assembled 6502 instructions.

```
EXAMPLE:  .DBYTE  $1234,1,-1
                  produces:  12 34 00 01 FF FF
          .WORD   $1234,1,-1
                  produces:  34 12 01 00 FF FF
```

---this page intentionally left blank---

CHAPTER 5: MACRO FACILITY

A MACRO DEFINITION is a series of source lines grouped together, given a name, and stored in memory. When the assembler encounters the corresponding name in the instruction (opcode, directive) column, the saved lines will be substituted for the Macro name and assembled. Effectively, this allows the user to define and then use new assembler instructions. Depending upon the code stored in its definition, a macro might be thought of as either an "extra" directive or a "new" opcode.

The process of finding a macro in the table when its name is used, and then assembling the code it was defined with, is called a MACRO EXPANSION. The unique facility of Macro Expansions is that they may have PARAMETERS passed to them. These parameters will be substituted for the "formal parameters" during the expansion of the Macro.

The use (expansion) of a Macro in a program requires that the Macro first be defined. To the set of directives already discussed in chapter 4, then, must be added two new directives used for defining new macros:

```
.MACRO
.ENDM
```

This chapter will first discuss these two directives, show how to invoke a macro (cause its expansion) and then examine the use of formal and calling parameters, including string parameters.

Section 5.1

directive: .ENDM

purpose: end the definition of a macro

usage:
```
.ENDM
```

usage note: generally, the .ENDM directive should not be labelled.

This directive is used solely to terminate the definition of a macro. When invoking a macro, do NOT use this directive. Basically, the concept of macros requires that all source lines between the .MACRO directive and the .ENDM directive be stored in a special section of memory (the macro table). Thus, encountering an improperly paired .ENDM directive is considered a severe assembly error. See the description of .MACRO for further information.

Section 5.2
---

directive: .MACRO

purpose: initiates a macro definition

usage: .MACRO macroname

usage note: "macroname" may be any valid MAC/65 label. It MAY be the same name as a program label (without conflict).

The .MACRO directive will cause the lines following to be read and stored under the Macro name of "macroname". The definition is terminated with the .ENDM directive.

All instructions except another .MACRO directive are valid Macro source lines. A Macro definition can NOT contain another Macro definition.

A simple example of a MACRO DEFINITION:

```
10    .MACRO PUSHXY ; The name of this Macro is "PUSHXY"
11 ; When this Macro is used (expanded), the following
12 ; instructions will be substituted for "PUSHXY"
13 ; and then assembled.
14    TXA
15    PHA
16    TYA
18    PHA
19    .ENDM      ; The terminator for "PUSHXY"
```

SPECIAL NOTE: ALL labels used within a macro are assumed to be local to that macro. MAC/65 accomplishes this by performing a "third pass" of the assembly during macro expansions. Thus, a label defined within a macro expansion is available to code which follows the macro; but another expansion of the same macro with the same label will reset the labels value. The action is similar to the ".=" directive, except that forward references to internal macro labels ARE legal.

An example follows, on the next page.

EXAMPLE:

```
20  .MACRO MOVE6
21  LDX #5
22 LOOP
23  LDA FROM,X
24  STA TO,X
25  DEX
26  BPL LOOP
27  .ENDM
```

The label "LOOP" is local to this macro usage, and yet it may (if needed) be referenced outside the macro expansion (although not in another macro expansion). (Note that if a macro label is only defined once by a single macro usage, the effect is the same as if the label were defined outside any macro.) Although the .LOCAL-produced local regions may be used by and with macros, the user is limited to a maximum of 62 local regions. No such restriction applies to the number of possible local usages of a label in a macro expansion.

5.3 MACRO EXPANSION, PART 1

As stated above, a macro is expanded when it is used. And the "use" of a macro is simplicity itself.

To invoke (use, expand--all equivalent words) a macro, simply place its name in the opcode/directive field of an assembler line. Remember, though, that macros MUST be defined before they can be used.

For example, to invoke the two macros defined in examples in the previous section (5.2), one could simply type them in as shown and then enter and assemble:

```
EXAMPLE:
        2000 ALABEL PUSHXY
        2010 ; and pushxy generates the code
        2020 ;    TXA   PHA   TYA   PHA
        2030 ;
        2040   MOVE6
        2050 ;    similarly, MOVE6 is used
        2060   JMP LOOP
        2070 ;    and LOOP refers to the label
        2080 ;    defined in the MOVE6 macro
        ...
```

Note that the use of a label on the macro invocation is optional. The label is assigned the current value of the location counter and is not dependent upon the contents of the macro at all.

There are many more "tricks" and features usable with macros, but we will continue this discussion after an examination of macro parameters as used in a macro definition.

5.4 MACRO PARAMETERS

Macro parameters can be of two types: expressions (which are evaluated as 16 bit words) or strings. The parameters are passed via the macro expansion (invocation, use, etc.) and are stacked in memory in the order of occurance. A maximum of 63 parameters can be stacked by a macro expansion, including expansions within expansions.

However, before a parameter can be used in an expansion, there must be a way of accessing it in the MACRO DEFINITION. Parameters are referenced in a macro definition by the character "%" for expressions and the characters "%$" for strings. The value following the character refers to the actual parameter number.

SPECIAL NOTE: The parameter number can be represented by a decimal number (e.g., %2) or may be a label enclosed by parentheses (e.g., %$(LABEL)). Of course, strings may be similarly referenced, as in %$(INDEX) or %$1.

Examples:

```
10      LDA     # >%1 ; get the high byte of parameter 1.
15      CMP     (%11 ,X) ; yes, that really is number 11.
20      .BYTE   %2-1    ; value of parameter 2 less 1.
        NOTE:    the above is NOT equivalent to using
                 parameter %1.  Parameter substitution
                 has highest precedence!

25 SYMBOL .= SYMBOL + 1
30      LDX     # -%(SYMBOL) ; see the power available?

40      .BYTE   %$1,%$2,0  ; string parameters, ending 0.
```

Remember, in theory the parameters are numbered from 1 to 63. In reality, the TOTAL number of parameters in use by all active (nested) macro expansions cannot exceed 63. This does NOT mean that you can have only 63 parameter references in your macro DEFINITIONS. The limit only applies at invocation time, and even then only to nested (not sequential) macro usages.

SPECIAL NOTE: In addition to the "conventional" parameters, referred to by number, parameter zero (%Ø) has a special meaning to MAC/65. Parameter zero allows the user to access the actual NUMBER of real parameters passed to a macro EXPANSION.

This feature allows the user to set default parameters within the Macro expansion, or test for the proper number of parameters in an expansion, or more. The following example illustrates a possible use of %Ø and shows usage of ordinary parameters as well.

EXAMPLE:

```
10   .MACRO BUMP
11 ;
12 ; This macro will increment the specified word
13 ;
14 ; The calling format is:
15 ;           BUMP  address [ ,increment ].
16 ; If increment is not given, 1 is assumed
17 ;
18   .IF %Ø=Ø .OR %Ø>2
19     .ERROR "BUMP:  Wrong number of parameters"
20   .ELSE
21 ;
22 ; this is only done if 1 or 2 parameters
23 ;
24     .IF %Ø>1  ; did user specify "increment" ?
25 ; this is assembled if user gave two parameters
26     LDA %1    ; add "increment" to "address".
27     CLC
28     ADC # <%2 ; low byte of the increment
29     STA %1    ; low byte of result
30     LDA %1 +1 ; high byte of location
31     ADC # >%2 ; add in high byte of increment
32     STA %1 +1 ; and store rest of result
33 ;
34     .ELSE
35 ; this is assembled if only one parameter given
36     INC %1    ; just increment by 1.
37     BNE SKIPHI ; implicitly local label
38     INC %1 +1 ; must also increment high byte
39 SKIPHI
40     .ENDIF    ; matches the .IF %Ø>1  (line 24)
41   .ENDIF     ; matches the .IF of line 18
42   .ENDM      ; terminator.
```

5.5 MACRO EXPANSION, PART 2

We have shown how macro definitions may include specifications of particular parameters (the specifications might also be called "formal parameters"). This section will show how to pass actual parameters (equivalently "value parameters", "calling parameters", etc.) to the definition.

The concept is simple: on the same line as the macro invocation (by use of its name, of course) and following the macro's name, the user may place expressions (or strings, see section 5.6). MAC/65 simply assigns each of these values a number, from 1 to 63, and then, during the macro expansion, replaces the formal parameters (%1, %2, %(label), etc.) with the corresponding values.

Does that sound too complicated? Internally, it is. Externally, it is as easy as this:

EXAMPLE:

Assume that the BUMP macro has been defined (as above, section 5.4), then the user may invoke it as needed, thus:

```
100 ALABEL BUMP A.LOCATION
110 INCR .= 7
120   BUMP A.LOCATION,3
130   BUMP A.LOCATION,INCR-2
140   BUMP
150   BUMP A.LOCATION,INCR,7
160 A.LOCATION .WORD 0
      note: lines 140 and 150 will each cause the
            BUMP error to be invoked and printed
```

Of course, you can also do silly things, which will no doubt produce some pretty horrible (and hard to debug) code:

```
170   BUMP INCR,A.LOCATION
       will try to increment address 7 by something
180   BUMP PORT5
       assuming that PORT5 is some hardware port,
       strange and wonderful things could happen
```

5.6 MACRO STRINGS

String parameters are represented in a macro definition by the characters "%$". All numeric parameters have a string counterpart, not all of which are useful. All string parameters have a numeric counterpart (their length).

As a special case, %$Ø always returns the macro NAME.

The following table shows the various string and numeric values returned for a given parameter:

As appears in Macro call:	string returned (in quotes):	numeric value returned:
"A String 1 2 3"	"A String 1 2 3"	length of string
NUMERICSYMBOL	"NUMERICSYMBOL"	value of label
SYMBOL+1	"SYMBOL"	value of expr
%$4	the string of parameter 4	value of orginal
(above would be used by a macro calling another macro)		
-LABEL	"LABEL"	value of expr
GEORGE*HARRY+PETE	undefined	value of expr
.DEF CIO	"CIO"	value of expr
2 + 2 * 65	undefined	value of expr

A Macro string example:

```
10   .MACRO PRINT
11 ;
12 ; This Macro will print the specified string,
13 ; parameter 1, but if no parameter string is
14 ; passed,  only an EOL will be printed.
15 ;
16 ; The calling format is:  PRINT  [ string ]
17 ;
18   .IF  %0 = 1 ; is there a string to print?
19   JMP PASTSTR ; yes, jump over string storage
20 STRING .BYTE %$1,EOL ; put string here.
21 ;
22 PASTSTR
23   LDX #>STRING ; get string address into X&Y
24   LDY #<STRING ; for JSR to 'print string'
25   JSR   STRINGOUT
26  .ELSE
27 ;   no string...just print an EOL
28   LDA   #EOL
29   JSR   CHAROUT
30 ;
31   .ENDIF
32   .ENDM          ; terminator.
```

To invoke this macro, then, the following calls would be appropriate:

```
100    PRINT "this is a string"
110    PRINT
120    PRINT MESSAGE
```

Line 120 is strange: The macro facility assumes that "MESSAGE" is a string (because of its usage), and so will print it exactly as if it had been placed in quotes. However, if the label MESSAGE is not defined elsewhere, the line will also generate an "Undefined Label" error. Generally, we do not suggest using this form. Use the quoted string instead.

5.7 SOME MACRO HINTS

Each person will soon develop his/her own style of writing macros, but there are certain common sense rules that we all should heed.

A. When a macro is defined, its entire definition must be stored in memory (in a macro table). Since memory space is obviously finite, it is a good idea to keep macros as short as possible. One way to do this is to avoid putting comments (remarks) within the body of the macro. If you do document your macros (and we hope you do), place the comments in the file BEFORE the .MACRO directive. The assembler will then do nothing at all with them and they will occupy no additional space.

B. Don't use a caller's macro parameter unless you are sure that it is there. Using a parameter that the caller left out will produce a MACRO PARAMETER error. Depending upon the macro definition, this may or may not also produce undesired results. An example of unsafe coding:

```
.IF %Ø>1 .OR %2=Ø
  .WORD %1
.ENDIF
```

The danger here occurs if the caller invokes the macro with only one parameter. Since %2 is non-existent (and hence undefined), the sub-expression "%2=Ø" is indeed true and the effect of "%Ø>1" is nullified. Of course, the lack of parameter 2 will produce a "PARAMETER ERROR", but it will already be too late. A better coding of the above would be:

```
.IF %Ø>1
  .IF %2<>Ø
    .WORD %1
  .ENDIF
.ENDIF
```

C. Even though labels defined within macros are local to each invocation, they are still "visible" outside the macro(s). Thus, it might be a good idea to have a special form for labels defined in macros and avoid that form outside macros. The macro library supplied with MAC/65 uses labels beginning with "@" as local labels to macros.

CAUTION: You should NOT define a label beginning with a question mark inside a macro. Neither should you use a .LOCAL directive within a macro. (You may USE labels that start with question marks, so long as you don't DEFINE them within the macro.)

5.8 A COMPLEX MACRO EXAMPLE

The following set of macros is designed to demonstrate several of the points made in the preceding sections. Aside from that, though, it is a good, usable macro set. Study it carefully, please. (The line numbers are omitted for the sake of brevity. Any numbers will do, of course.)

```
;
; the first macro, "@CH", is designed to load an
; IOCB pointer into the X register.  If passed a
; value from Ø to 7, it assumes it to be a constant
; (immediate) channel number.  If passed any other
; value, it assumes it to be a memory location which
; contains the channel number.
;
; NOTE that these comments are outside the body of
; the macro, thus saving valuable table space.
;
       .MACRO @CH
       .IF  %1>7 ; where is channel number?
       LDA  %1   ; channel # is in memory cell
       ASLA      ; so load it and
       ASLA      ; multiply it
       ASLA      ; 16 via
       ASLA      ; these shifts
       TAX       ; then move it to X register
       .ELSE
       LDX #%1*16 ; channel # times 16 goes in X
       .ENDIF
       .ENDM
;
; this next macro, "@CV", is designed to load a
; Constant or Value into the A register.  If
; passed a value from Ø to 255, it assumes it
; to be a constant (immediate) value.  If passed
; any other value, it assumes it to be a memory
; location (non-zero page).
;
       .MACRO @CV
       .IF  %1<256 ; is this a constant value?
       LDA  #%1    ; yes...so load it immediately
       .ELSE
       LDA  %1     ; no...so get it from memory
       .ENDIF
       .ENDM
```

```
;
; The third macro is "@FL", designed to establish
; a filespec.  If passed a literal string, @FL
; will generate the string in line, jumping around
; it, and place its address in the IOCB pointed to
; by the X register.  If passed a non-zero page
; label, @FL assumes it to be the label of a valid
; filespec string and uses it instead.
;
       .MACRO  @FL
       .IF  %1<256 ; is this a literal string?
       JMP  *+%1+4 ; yes...so jump around the string
@F     .BYTE %$1,Ø ; ...and store the string here
       LDA  #<@F   ; then get address of the string
       STA  ICBADR,X ; put in IOCB's address field
       LDA  #>@F   ; also high byte of address
       STA  ICBADR+1,X
       .ELSE
       LDA  #<%1   ; not a literal string
       STA  ICBADR,X ; but still get its address
       LDA  #>%1   ; (both bytes)
       STA  ICBADR+1,X ; to IOCB's address field
       .ENDIF
       .ENDM
```

```
;
; The main macro here is "XIO", a macro to
; implement a simulation of BASIC's XIO command.
; The general syntax of the usage of this macro is:
;   XIO command,channel [,aux1,aux2] [,filespec]
;
; where channel may be a constant from Ø to 7
;       or a memory location.
; where command, aux1, and aux2 may be a constant
;       from Ø to 255 or a non-zero page location
; where filespec may be a literal string or
;        a non-zero page location
; if aux1 and aux2 are omitted, they are assumed
;       to be zero (you may not omit aux2 only)
; if the filespec is omitted, it is assumed to
;       be "S:"
;
        .MACRO  XIO
        .IF  %Ø<2 .OR %Ø>5 ; just checking
        .ERROR "XIO: wrong number of parameters"
        .ELSE
          @CH %2    ; process the channel number
          @CV %1    ; and the XIO command number
          STA ICCOM,X ; ...putting command # in IOCB
          .IF  %Ø>=4 ; 4 or 5 arguments given?
            @CV %3    ; yes...so process
            STA ICAUX1,X  ; aux 1
            @CV %4
            STA ICAUX2,X  ; and aux 2
           .ELSE      ; 2 or 3 arguments given
            LDA #Ø  ; so assume value of zero
            STA ICAUX1,X ; for aux 1
            STA ICAUX2,X ; and aux 2
          .ENDIF
          .IF  %Ø=2 .OR %Ø=4 ; was filename given?
            @FL  "S:" ; no...assume name is "S:"
          .ELSE        ; but if yes...
@FPTR     .=  %Ø       ; get parameter number of name
            @FL  %$(@FPTR)  ; and process it
          .ENDIF
          JSR  CIO     ; call the OS
        .ENDIF
        .ENDM
```

Did you follow all that? The trick is that, the way "XIO" is specified, it is legal to pass it 2, 3, 4, or 5 arguments; but each of those numbers represents a unique combination of parameters, to wit:

```
XIO     command,channel
XIO     command,channel,filespec
XIO     command,channel,aux1,aux2
XIO     command,channel,aux1,aux2,filespec
```

This is not a trivial macro example. Perhaps you will not have occasion to write something so complex. But MAC/65 provides the tools to do many things if you need them.

SPECIAL NOTE: Appendix B contains a fairly complete set of I/O macros which you may type in and use.

ALSO: You may inquire about the availability of the OSS MAC/65 Programmers' Aid Disk, which should include all the macros in Appendix B and many more.

CHAPTER 6: COMPATIBILITY

There are many different 6502 assemblers available, and it seems that each has a few foibles, bugs, or whatever that are uniquely its own (and, of course, they are called "features" by their promoters). Well, MAC/65 is no different.

This chapter is devoted to telling you of some of the things to watch out for when converting from another 6502 assembler to MAC/65. We will restrict ourselves to such things as directives and operators. We will NOT go into a discussion of how to convert the actual 6502 opcodes (equivalently: instructions, mnemonics, etc.). We consider it mandatory that any good 6502 assembler will follow the MOS Technology standard in this regard.

Example: We know of some antique 6502 assemblers that specify the various addressing modes via special opcodes. Thus the conventional "LDA #3" becomes "LDAIMM 3" and "LDA (ZIP),Y" becomes "LDAIY ZIP". Unfortunately, there was never any standard established for such distortions, so we shall ignore them as antique and outmoded. In any case, unless you are entering a program out of an older magazine, you are unlikely to run into one of these strange beasts.

The rest of this chapter pays homage to our birthright. MAC/65 is a direct descendant of the Atari assembler/editor cartridge (via EASMD). As much as possible, we have tried to keep MAC/65 compatible with the cartridge. Unfortunately, in the interest of providing a more powerful tool, a few things had to be changed. The next section of this chapter, then, enumerates these changes.

6.1 ATARI'S ASSEMBLER/EDITOR CARTRIDGE

This section presents all known functional differences between the Atari cartidge and MAC/65. Obviously, MAC/65 also has many more features not enumerated here, but they will not impact the transferrance of code originally designed for the cartridge (or, for that matter, EASMD).

6.1.1 .OPT OBJ / NOOBJ

By default, the Atari cartridge produces object code, even when the destination of the object is RAM memory. This is a dangerous practice, at best: it is too easy to make a mistake in a program and write over DOS, the user's source, the screen memory, or even (horror of horrors) some of the hardware registers.

MAC/65 makes a special case of object in memory: you don't get it unless you ask for it. You MUST have a ".OPT OBJ" directive before the code to be generated or the code will not be produced.

6.1.2 OPERATOR PRECEDENCE

The Atari cartridge assigns no precedence to arithmetic operators. MAC/65 uses a precedence similar to BASIC's. Most of the time, this causes no problems; but watch out for mixed expressions.

```
Example:       LDA #LABEL-3/256
   seen as     LDA #{LABEL-3} / 256 by the cartridge
   seen as     LDA #LABEL - {3/256} by MAC/65
```

6.1.3 THE .IF DIRECTIVE

The implementation of .IF in the cartridge is clumsy and unusable. MAC/65's implementation is more conventional and much more powerful. Rather than try to offer a long example here, we will simply refer you to the appropriate sections of the two manuals.

6.1.4 ZERO PAGE FORWARD REFERENCES

MAC/65 can not properly assemble a forward reference to a zero page label (usually, you will get a PHASE ERROR). The Atari cartridge generally can, but it has other limitations on addressing modes which MAC/65 does not suffer under.

You can usually avoid phase errors simply by moving your equates for zero page locations to the head of your assembled code.

CHAPTER 7: ADDED 65CØ2 INSTRUCTIONS

MAC/65, as originally produced, supported the "standard" 65Ø2 instruction set as well as the directives and addressing mode designators recommended by MOS Technology (the originators of the 65Ø2 chip).

This version of MAC/65 supports all features of the original version along with added support for one of the more popular enhanced versions of the 65Ø2 chip. In particular, MAC/65 supports all new instructions and addressing modes available on the 65CØ2 chip as produced by NCR Corporation.

We describe here the primary added addressing mode, the instructions with variants added, and the completely new instructions.

But before we start, we should note that these instructions will only work properly on your computer if you have installed an NCR 65CØ2 in place of the 65Ø2 which came in the machine as purchased. Also, remember that a program using these instructions may work great in your machine. It will not work properly in your friend's machine unless he/she also installs a 65CØ2.

7.1 A Major Added Addressing Mode

The standard 6502 chip supports two forms of indirect addressing for what might be considered its primary instructions. The forms appear in assembly listings as

```
lda   (indirect,X)
        and
lda   (indirect),Y
```

(where "lda" is only one of several valid mnemonics that can be used with these addressing modes).

The latter of these modes, often referred to as the "indirect-Y" mode, is perhaps the most useful and flexible of all 6502 addressing modes. And, yet, it suffers from one flaw: it ties up two registers (A and Y). And, as importantly, probably a full 50% or more of the time the Y-register is loaded with zero before instructions in this mode are executed.

The NCR 65C02 instruction set as supported by MAC/65 provides a help here: you may code instructions allowing Indirect-Y addressing in "Indirect" mode as well. With Indirect mode, the assembler format is simply

```
lda   (indirect)
```

where, as with Indirect-Y, the indirect location must be in zero page.

Generally, the effect of using this instruction will be the same as coding the sequence:

```
LDY   #0
lda   (indirect),Y
```

EXCEPTING that the Y-register remains intact and untouched and may be used for other purposes.

The following, then, are ALL of the 65C02 instructions which allow and support this new addressing mode:

```
ADC (indirect)          ; ADd with Carry
AND (indirect)          ; bitwise AND
CMP (indirect)          ; compare with A-reg
EOR (indirect)          ; Exclusive OR
LDA (indirect)          ; LoaD the A-register
ORA (indirect)          ; inclusive OR
SBC (indirect)          ; SuBtract with Carry
STA (indirect)          ; STore the A-register
```

REMINDER: while the "indirect" location may be any zero page location, you should probably restrict yourself to the available locations documented in the DDT manual.

7.2 Minor Variations on 6502 Instructions

The "BIT" instruction has added two new addressing modes, and "JMP" has added one new mode. They are described here individually:

Original allowed forms of 6502 BIT instruction were:

```
BIT  absolute
BIT  zeropage
```

New 65C02 forms available are:

```
BIT  absolute,X
BIT  zeropage,X
```

The ability to use the X register as in index with the BIT instruction greatly enhances its power for testing tables, etc. The "indexed-x" address modes function as they do for other 6502 instructions (e.g.,LDA and CMP).

Original allowed forms of 6502 JMP instruction were:

```
JMP  absolute
JMP  (indirect)
```

New 65C02 form available is:

```
JMP  (indirect,X)
```

Note that the JMP instruction alone in both the 6502 and 65C02 instructions sets uses an absolute (i.e., 16 bit, 2 byte) address for its indirect value. The new "indirect-X" form is no different: the location specified as the indirect address may be anywhere in memory.

This "indirect-X" address mode is unique and new. Its effect is as follows: add the contents of the X-register to the ADDRESS (not the contents) specified by the given indirect address; use the result as the address of the true operand for this instruction; JuMP to the address contained in the word-sized location accessed via the true operand.

An example is in order:

```
TABLE  .WORD  SUB1,SUB2,SUB3
        ...
        LDA      value  ; assume that "value"
                        ; contains 0,1, or 2
        ASL A           ; double the value
        TAX             ; ...to X-register
        JMP (TABLE,X)   ; and go to SUB1, SUB2,
                        ; SUB3 depending on "value"
```

7.3 ALL-NEW 65C02 Instructions

We detail here, in what we hope are logical groupings, the 65C02 instructions which are truly "new" to the 6502 world.

7.3.1 BRA

Mnemonic: BRA

Read as: BRAnch

Format: BRA addr
where addr must be in the range *-126 to *+129 (* is the current value of the location counter)

Comments: BRA joins the Branch family (BNE, BEQ, BMI, etc.) and adds the powerful capability of ALWAYS branching. It thus becomes equivalent to a JMP instruction with the advantage that it occupies one less byte in memory and is inherently relocatable. Its address range is restricted in a fashion identical with the other members of the "branch" family.

7.3.2 DEA and INA

Mnemonics: DEA
INA

Read as: DEcrement Accumulator
INcrement Accumulator

Formats: DEA
INA

Comments: These simple instructions add a capability long lacking in the 6502. Until now, if you wished to change the contents of the accumulator by one, you had to either use TAX/INX/TXA (or something similar) or CLC/ADC (or SEC/SBC), three byte substitutes for what should (and now is) a single byte instruction.

Processor status flags (i.e., N and Z), timings, etc., are all identical to the very similar INX/INY/DEX/DEY set of instructions.

7.3.3 PHX, PHY, PLX, and PLY

Mnemonics:
```
PHX
PHY
PLX
PLY
```

Read as:
```
PusH X onto CPU stack
PusH Y onto CPU stack
PulL X from CPU stack
PulL Y from CPU stack
```

Formats:
```
PHX
PHY
PLX
PLY
```

Comments: Again, these instructions are provided as short cuts for the cumbersome sequences necessary on the standard 6502. As an example, PHX can replace a sequence of instructions as complex as this:

```
STA temp
TXA
PHA
LDA temp
```

By giving you direct access to the stack from the X and Y registers, it is possible and desirable to right more compact and more relocatable code. Processor status flag usage, timings, etc., are identical to the very similar PHA and PLA instructions.

7.3.4 STZ

Mnemonic: STZ

Read As: STore Zero

Formats:
```
STZ  absolute
STZ  absolute,X
STZ  zeropage
STZ  zeropage,X
```

Comments: Yet another short cut, STZ simply replaces the sequence

```
LDA #Ø
STA address
```

with the difference that it does not affect the contents of the A register. In fact, to properly simulate this instruction on an ordinary 65Ø2, the following code would be needed in the general case:

```
PHA
LDA #Ø
STA address
PLA
```

7.3.5 TRB and TSB

Mnemonics:
```
TRB
TSB
```

Read As:
```
Test and Reset Bits
Test and Set Bits
```

Formats:
```
TRB  absolute
TRB  zeropage
TSB  absolute
TSB  zeropage
```

Comments: These instructions have many uses, not the least of which would be synchronization of background and foreground (interrupt-driven) routines. In boolean terms, the instructions might be thought of thus:

```
TRB:  Memory := (Not A) and Memory
TSB:  Memory := A or Memory
```

In words, we might describe the operation of these instructions as follows:

For TRB: The complement of the contents of the Accumulator is bit-wise AND-ed with the contents of the memory cell addressed by this instruction (either an absolute or zero-page location). The result of this AND-ing is placed back in the addressed memory cell.

For TSB: The contents of the Accumulator is bit-wise OR-ed with the contents of the memory cell addressed by this instruction. The result of this OR-ing is placed back in the addressed memory cell.

If the result of the AND-ing or OR-ing is zero, the Zero processor status flag is set. The N and V flags are set to the contents of bits 6 and 7 (similar to the usage and results of the BIT instruction) of the addressed memory cell as those contents were BEFORE the bit-wise operation took place.

Examples:

```
FLAG .BYTE 3
TEST .BYTE $FF
     ...
     LDA #$FF
     TRB FLAG ; resets all bits!
     ...
     LDA #Ø
     TSB TEST ; just tests value
```

+

+

---this page intentionally left blank--

+

+

CHAPTER 8: PROGRAMMING TECHNIQUES WITH MAC/65

This chapter will present you with a couple of hints about how to use MAC/65 to more advantage.

8.1 Memory Usage by MAC/65 and DDT

The following memory locations are used by MAC/65 and/or DDT for the purposes shown:

range of addresses	used by MAC/65	DDT	used for
$80-$AF	yes	yes	pointers and temporaries
$B0-$D3	yes	no	pointers and temporaries
$D4-$FF	yes	no	floating point registers, etc.
$100-$1FF	yes	yes	normal 6502 CPU stack
$3FD-$47F	no	yes	buffers and display area
$480-$57F	yes	yes	buffers and work area
$580-$67F	yes	no	input buffers, etc.
"size"	yes	*	program text, etc.

Note that "size" refers to the memory area delineated by the lowest and middle numbers displayed when the "SIZE" command is used from the MAC/65 editor. The * in DDT's column indicates that DDT saves MAC/65's zero page memory (and other, related, locations) in the area actually shown to be part of the "size" memory.

The worst implication of the memory map above (especially for Atari BASIC users) is that page 6 is NOT completely available to you. Since many magazine articles assume that page 6 is available, they will not run AS IS under MAC/65 and DDT. But see the next section for methods to use if you MUST use page 6.

8.2 Assembling With An Offset: .SET 6

In Section 4.19, we noted that the assembler directive ".SET 6,value" could be used to specify an additive offset for the storage address vis-a-vis the location counter address. In this section, we present a method for using this capability in a practical sense.

Let us assume that we wish to assemble a small program which will reside in page 6 ($600 through $6FF). The program which we will assemble is presented here:

```
  10          *= $600
  20 COLOR4   =  $2C8
  30 ;
  40 START
  50          PLA         ; remove count of parameters
  60          CMP #0      ; any parameters?
  70          BEQ *       ; if yes, loop forever
  80          LDA COLOR4  ; get current background color
  90          CLC
 100          ADC #$10    ; change to next hue
 110          STA COLOR4  ; ...by changing shadow reg
 120          INC COUNT   ; and count the number of times
 130          RTS
 140 COUNT    .BYTE 0     ; just a simple counter
 150          .END
```

If you assemble this routine, you shoud get an error free assembly. (And those of you who are BASIC users will recognize this as a routine callable from Atari BASIC, thanks to the PLA and check on number of parameters at the beginning.)

But it is designed to reside in page 6. What can we do? Answer: simply add the following two lines to the listing:

```
  12          .OPT OBJ    ; we do want object code
  14          .SET 6,$3000 ; and we will offset
```

Now, if you assemble this code, you will notice that the addresses shown start at $3600. And, indeed, the assembler is placing the code in memory at the addresses shown. But look at line 120. Notice that the object code generated does NOT show that location $3612 is being incremented! Instead, location $0612 is affected. Also note that in the symbol table listing START is shown to be at location $600 and COUNT at $612.

Now use the "DDT" command to enter DDT. From DDT, enter the command

```
M 3600060000080 [RETURN]
```

which will move $80 (128) bytes from location $3600 to location $600. Use the command

```
* 0600 [RETURN]
```

to view the contents of locations $600 and beyond. Use the up and down arrows (remember, WITHOUT pushing CTRL) to view the code. Lo and behold, your code has been successfully deposited where you wanted it, waiting for you to debug.

Some final notes on this subject: MAC/65 will generate this "offset" kind of code either directly to memory (as we did here) or to an object file (on disk, presumably). When the file is reloaded (via MAC's BLOAD command or via some load command from the DOS you are using), it will be loaded at the address shown in the listing. It is your responsibility to then somehow move it to the desired location. The technique is not necessarily easy, but using these methods you can overwrite DOS or even produce code designed to run in the cartridge space. In the latter case, you may wish to use a negative offset with .SET 6. This is perfectly legal and reasonable.

8.3 Making MAC/65 Even Faster

If you .INCLUDE a file consisting ONLY of equates and/or macro definitions (NOT macro calls!), there is a technique you can use which will speed up assembly somewhat.

In particular, since equates need be made only once and macros need be only defined once, there is no reason to read such .INCLUDED files on pass two. The following code shows a workable technique:

```
        *= Ø
PASS    .= PASS+1  ; do this only once per assembly
        .IF PASS=1
        .INCLUDE #D:equatesfile
        .ENDIF
        *= beginning
```

Why this works: Normally, an undefined label has a value of zero. The ".=" directive, however, causes a mildly strange thing to happen: an undefined label used on the right side of ".=" takes on the current value of the location counter. Hence the need for the " *= Ø" line at the beginning of the above example.

In any case, thanks to this mechanism, the first time the second line is assembled (in pass 1), PASS takes on a value of 1 (of course, the line also generates an "undefined label" error, but such errors are not printed in pass 1). The next time it is assembled, PASS receives a value of 2. Simple and neat.

Note that if the ".=" used in the second line above is placed ahead of any "*=" (or ".ORG") lines, then the first line shown is not needed, since the location counter is assumed to start at zero unless told otherwise.

Appendix A: System Equates

We present here a listing of certain system locations which we find useful and necessary when programming on the Atari Computer.

Many of the equates shown here are noted as applying to DOS XL. Generally, if you are working with system resources (such as IOCB's and CIO and such), the equates will be identical for Atari DOS. We have tried to specially mark the locations which apply only to DOS XL (especially batch execution and the command line).

Some of the labels on these equates may vary slightly from those used by Atari (in the operating system listings) or in published books (such as "Mapping The Atari", from Compute! books). The differences are minimal (e.g., ICAX1 instead of ICAUX1).

You may type in this entire listing and SAVE the result to disk or tape. If you save it to disk, you may later .INCLUDE it for use by your program(s). If you save it to disk, you will have to merge it with (or append it to) your programs.

You may also simply use this listing as a reference, typing in only the equated labels that your program actually uses.

(The listing begins on the next page.)

```
 1000      .PAGE "OSS SYSTEM EQUATES FOR ATARI"
1010 ;
1020 ;  Recommended File Name:  SYSEQU.M65
1030 ;
1040 ;
1050 ; I/O CONTROL BLOCK EQUATES
1060 ;
1065 SAVEPC = *         ; SAVE CURRENT ORG
1067 ;
1070      *=  $0340     ;START OF SYSTEM IOCBS
1075 IOCB
1080 ;
1090 ICHID .DS 1        ;DEVICE HANDLER IS (SET BY OS)
1100 ICDNO .DS 1        ;DEVICE NUMBER (SET BY OS)
1110 ICCOM .DS 1        ;I/O COMMAND
1120 ICSTA .DS 1        ;I/O STATUS
1130 ICBADR .DS 2       ;BUFFER ADDRESS
1140 ICPUT .DS 2        ;DH PUT ROUTINE (ADR-1)
1150 ICBLEN .DS 2       ;BUFFER LENGTH
1160 ICAUX1 .DS 1       ;AUX 1
1170 ICAUX2 .DS 1       ;AUX 2
1180 ICAUX3 .DS 1       ;AUX 3
1190 ICAUX4 .DS 1       ;AUX 4
1200 ICAUX5 .DS 1       ;AUX 5
1210 ICAUX6 .DS 1       ;AUX 6
1220 ;
1230 IOCBLEN = *-IOCB ;LENGTH OF ONE IOCB
1240 ;
1250 ; IOCB COMMAND VALUE EQUATES
1260 ;
1270 COPN =  3          ;OPEN
1280 CGBINR = 7         ;GET BINARY RECORD
1290 CGTXTR = 5         ;GET TEXT RECORD
1300 CPBINR = 11        ;PUT BINARY RECORD
1310 CPTXTR = 9         ;PUT TEXT RECORD
1320 CCLOSE = 12        ;CLOSE
1330 CSTAT = 13         ;GET STATUS
1340 ;
1350 ; DEVICE DEPENDENT COMMAND EQUATES FOR FILE MANAGER
1360 ;
1370 CREN =  32         ;RENAME
1380 CERA =  33         ;ERASE
1390 CPRO =  35         ;PROTECT
1400 CUNP =  36         ;UNPROTECT
1410 CPOINT = 37        ;POINT
1420 CNOTE = 38         ;NOTE
1430 ;
1440 ; AUX1 VALUES REQD FOR OPEN
1450 ;
1460 OPIN =  4          ;OPEN INPUT
1470 OPOUT = 8          ;OPEN OUTPUT
1480 OPUPD = 12         ;OPEN UPDATE
1490 OPAPND = 9         ;OPEN APPEND
1500 OPDIR = 6          ;OPEN DIRECTORY
1510 ;
```

```
1520      .PAGE
1530 ;
1540 ;     EXECUTE FLAG DEFINES
1550 ;
1560 EXCYES = $80     ; EXECUTE IN PROGRESS
1570 EXCSCR = $40     ; ECHO EXCUTE INPUT TO SCREEN
1580 EXCNEW = $10     ; EXECUTE START UP MODE
1590 EXCSUP = $20     ; COLD START EXEC FLAG
1600 ;
1610 ; MISC ADDRESS EQUATES
1620 ;
1630 CPALOC = $0A     ; POINTER TO CP
1640 WARMST = $08     ; WAR, START (0=COLD)
1650 MEMLO = $02E7    ; AVAIL MEM (LOW) PTR
1660 MEMTOP = $02E5   ; AVAIL MEM (HIGH) PTR
1670 APPMHI = $0E     ; UPPER LIMIT OF APPLICATION MEMORY
1680 INITADR = $02E2 ; ATARI LOAD/INIT ADR
1690 GOADR = $02E0    ; ATARI LOAD/GO ADR
1700 CARTLOC = $BFFA ; CARTRIDGE RUN LOCATION
1710 CIO =    $E456   ;CIO ENTRY ADR
1720 EOL =    $9B     ; END OF LINE CHAR
1730 ;
1740 ;  CP FUNCTION AND VALUE DISPLACEMSNT
1750 ;      (INDIRECT THROUGH CPALOC)
1760 ;            IE. (CPALOC),Y
1770 ;
1780 CPGNFN = 3       ; GET NEXT FILE NAME
1790 CPDFDV = $07     ; DEFAULT DRIVE (3 BYTES)
1800 CPBUFP = $0A     ; CMD BUFF NEXT CHAR POINTR (1 BYTE)
1810 CPEXFL = $0B     ; EXECUTE FLAG
1820 CPEXFN = $0C     ; EXECUTE FILE NAME (16 BYTES)
1830 CPEXNP = $1C     ; EXECUTE NOTE/POINT VALUES
1840 CPFNAM = $21     ; FILENAME BUFFER
1850 RUNLOC = $3D     ; CP/A LOAD/RUN ADR
1860 CPCMDB = $3F     ; COMMAND BUFFER (60 BYTES)
1870 CPCMDGO = $F3
1880 ;
1890      *=  SAVEPC  ; RESTORE PC
1900 ;
```

+

+

+

---this page intentionally left blank---

+

+

+

Appendix B: Some Useful Macros

In the pages which follow, we present the listings of several macros. These macros are designed to make it easy for you to perform Input/Output operations. If you type all of them in exactly as shown, you will have a useful macro library.

We suggest that you type them in and then SAVE them (to disk or tape). If you save them to disk, you can later use .INCLUDE to allow your program access to their ease and power. If you save them to tape, you will have to merge them with your program in memory in order to use them.

CAUTION: These macros use many of the equates given in the SYSTEM EQUATES listing of Appendix A. You may either .INCLUDE the entire set of equates as presented or simply type in the ones which these macros need. (You can find out which labels they need by assembling your program without the equates. The undefined labels will causes errors during the assembly.)

Before we present the listings, we present here a summary of each macro along with notations on how to use it. Remember, using a macro requires simply coding its name in the operator (mnemonic) field of a line along with any parameters in the operand field(s).

The macros are presented here in expected order of usage:

OPEN chan,aux1,aux2,filename
: Opens the given filename on the given channel using aux1 and aux2 as per OS/A+ specifications.

PRINT chan [,buffer [,length]]
: If no buffer given, prints just a CR on chan. If no length given, length assumed to be 255 or position of CR, whichever is smaller. Buffer may be literal string, in which case length is ignored if given.

INPUT chan,buffer [,length]
: If no length given, defaults to 255 bytes.

BGET chan,buffer,length
: Binary read, a la BASIC XL, of length number of bytes into the given buffer address.

BPUT chan,buffer,length
Binary write of length number of bytes from the given buffer address.

CLOSE chan
Closes the given file.

XIO command,chan [,aux1,aux2][,filename]
As described in chapter 5.

NOTES:

"chan" may be a literal channel number (Ø through 7) or a memory location containing a channel number (Ø through 7).

"aux1", "aux2", "length", and "command" may all be either literal numbers (Ø to 255) or memory locations.

"filename" may be either a literal string (e.g., "D:FILE1.DAT") or a memory location, the latter assumed to be the address of the start of the filename string.

Where memory locations are given instead of literals, they must be non-zero page locations which are defined BEFORE their usage in the macro(s). The following example will NOT work properly !! :

```
          PRINT 3,MESSAGE1  ; WRONG!
          ...
     MESSAGE1 .BYTE "This WON'T WORK !!! "
```

These macros are useful instruments, but they really are meant only as examples, to show you what you can do with MAC/65. Please feel free to study them and change them as you need.

(The listings start on the next page.)

```
1000      .TITLE "IOMAC.LIB -- OSS system I/O macros"
1010      .PAGE "   Support Macros"
1020        .IF .NOT .DEF IOCB
1030        .ERROR "You must include SYSEQU.M65 ahead of this!!"
1040        .ENDIF
1050 ;
1060 ; These macros are called by the actual I/O macros
1070 ; to perform the rudimentary register load functions.
1080 ;
1090 ;
1100 ; MACRO:  @CH
1110 ;
1120 ; Loads IOCB number (parameter 1) into X register.
1130 ;
1140 ; If parameter value is 0 to 7, immediate channel number
1150 ;   is assumed.
1160 ;
1170 ; If parameter value is > 7 then a memory location
1180 ;   is assumed to contain the channel number.
1190 ;
1200      .MACRO @CH
1210        .IF %1>7
1220        LDA %1
1230        ASL A
1240        ASL A
1250        ASL A
1260        ASL A
1270        TAX
1280        .ELSE
1290        LDX #%1*16
1300        .ENDIF
1310      .ENDM
1320 ;
1330 ;
1340 ; MACRO:  @CV
1350 ;
1360 ; Loads Constant or Value into accumultor (A-register)
1370 ;
1380 ; If value of parameter 1 is 0-255, @CV
1390 ; assumes it's an (immediate) constant.
1400 ;
1410 ; Otherwise the value is assumed to
1420 ; be a memory location (non-zero page).
1430 ;
1440 ;
1450 ;
1460      .MACRO @CV
1470        .IF %1<256
1480        LDA #%1
1490        .ELSE
1500        LDA %1
1510        .ENDIF
1520      .ENDM
1530 ;
1540 ;
1550 ;
```

```
1560 ;
1570 ; MACRO:  @FL
1580 ;
1590 ; @FL is used to establish a filespec (file name)
1600 ;
1610 ; If a literal string is passed, @FL will
1620 ; generate the string in line, jump
1630 ; around it, and place its address
1640 ; in the IOCB pointed to by the X-register.
1650 ;
1660 ; If a non-zero page label is passed
1670 ; the MACRO assumes it to be the label
1680 ; of a valid filespec and uses it instead.
1690 ;
1700 ;
1710 ;
1720      .MACRO @FL
1730        .IF %1<256
1740        JMP *+%1+4
1750 @F     .BYTE %$1,0
1760        LDA # <@F
1770        STA ICBADR,X
1780        LDA # >@F
1790        STA ICBADR+1,X
1800        .ELSE
1810        LDA # <%1
1820        STA ICBADR,X
1830        LDA # >%1
1840        STA ICBADR+1,X
1850        .ENDIF
1860      .ENDM
1865 ;
```

```
1870      .PAGE "   XIO macro"
1880 ;
1890 ; MACRO:  XIO
1900 ;
1910 ;  FORM:  XIO cmd,ch[,aux1,aux2][,filespec]
1920 ;
1930 ; ch is given as in the @CH macro
1940 ; cmd, aux1, aux2 are given as in the @CV macro
1950 ; filespec is given as in the @FL macro
1960 ;
1970 ; performs familiar XIO operations with/for OS/A+
1980 ;
1990 ; If aux1 is given, aux2 must also be given
2000 ; If aux1 and aux2 are omitted, they are set to zero
2010 ; If the filespec is omitted, "S:" is assumed
2020 ;
2030      .MACRO XIO
2040        .IF %0<2 .OR %0>5
2050        .ERROR "XIO: wrong number of arguments"
2060        .ELSE
2070         @CH  %2
2080         @CV  %1
2090        STA ICCOM,X ; COMMAND
2100          .IF %0>=4
2110           @CV  %3
2120          STA ICAUX1,X
2130           @CV  %4
2140          STA ICAUX2,X
2150          .ELSE
2160          LDA #0
2170          STA ICAUX1,X
2180          STA ICAUX2,X
2190          .ENDIF
2200          .IF %0=2 .OR %0=4
2210           @FL  "S:"
2220          .ELSE
2230 @@IO      .=  %0
2240           @FL  %$(@@IO)
2250          .ENDIF
2260        JSR CIO
2270        .ENDIF
2280      .ENDM
2285 ;
```

```
2290      .PAGE "   OPEN macro"
2300 ;
2310 ; MACRO:  OPEN
2320 ;
2330 ;  FORM:  OPEN ch,aux1,aux2,filespec
2340 ;
2350 ; ch is given as in the @CH macro
2360 ; aux1 and aux2 are given as in the @CV macro
2370 ; filespec is given as in the @FL macro
2380 ;
2390 ; will attempt to open the given file name on
2400 ; the given channel, using the open "modes"
2410 ; specified by aux1 and aux2
2420 ;
2430      .MACRO OPEN
2440        .IF %0<>4
2450        .ERROR "OPEN: wrong number of arguments"
2460        .ELSE
2470          .IF %4<256
2480           XIO  COPN,%1,%2,%3,%$4
2490          .ELSE
2500           XIO  COPN,%1,%2,%3,%4
2510          .ENDIF
2520        .ENDIF
2530      .ENDM
2535 ;
```

```
2540      .PAGE "   BGET and BPUT macros"
2550 ;
2560 ; MACROS: BGET and BPUT
2570 ;
2580 ;    FORM: BGET ch,buf,len
2590 ;          BPUT ch,buf,len
2600 ;
2610 ; ch is given as in the @CH macro
2620 ; len is ALWAYS assumed to be an immediate
2630 ;   and actual value...never a memory address
2640 ; buf must be the address of an appropriate
2650 ;   buffer in memory
2660 ;
2670 ; puts or gets length bytes to/from the
2680 ;   specified buffer, uses binary read/write
2690 ;
2700 ;
2710 ; first: a common macro
2720 ;
2730      .MACRO @GP
2740       @CH  %1
2750      LDA #%4
2760      STA ICCOM,X
2770      LDA # <%2
2780      STA ICBADR,X
2790      LDA # >%2
2800      STA ICBADR+1,X
2810      LDA # <%3
2820      STA ICBLEN,X
2830      LDA # >%3
2840      STA ICBLEN+1,X
2850      JSR CIO
2860      .ENDM
2870 ;
2880      .MACRO BGET
2890        .IF %0<>3
2900        .ERROR "BGET: wrong number of parameters"
2910        .ELSE
2920         @GP  %1,%2,%3,CGBINR
2930        .ENDIF
2940      .ENDM
2950 ;
2960      .MACRO BPUT
2970        .IF %0<>3
2980        .ERROR "BPUT: wrong number of parameters"
2990        .ELSE
3000         @GP  %1,%2,%3,CPBINR
3010        .ENDIF
3020      .ENDM
3030 ;
```

```
3040      .PAGE "   PRINT macro"
3050 ;
3060 ; MACRO:  PRINT
3070 ;
3080 ;  FORM:  PRINT ch[,buffer[,length]]
3090 ;
3100 ; ch is as given in @CH macro
3110 ; if no buffer, prints just a RETURN
3120 ; if no length given, 255 assumed
3130 ;
3140 ; used to print text.  To print text without RETURN,
3150 ; length must be given.  See OS/A+ manual
3160 ;
3170 ; EXCEPTION: second parameter may be a literal
3180 ;  string (e.g., PRINT 0,"test"), in which
3190 ;  case the length (if given) is ignored.
3200 ;
3210      .MACRO PRINT
3220        .IF %0<1 .OR %0>3
3230        .ERROR "PRINT: wrong number of parameters"
3240        .ELSE
3250          .IF %0>1
3260            .IF %2<128
3270            JMP *+4+%2
3280 @IO          .BYTE %$2,$9B
3290              @GP  %1,@IO,%2+1,CPTXTR
3300            .ELSE
3310              .IF %0=2
3320                @GP  %1,%2,255,CPTXTR
3330              .ELSE
3340                @GP  %1,%2,%3,CPTXTR
3350              .ENDIF
3360            .ENDIF
3370          .ELSE
3380          JMP *+4
3390 @IO      .BYTE $9B
3400            @GP  %1,@IO,1,CPTXTR
3410          .ENDIF
3420        .ENDIF
3430      .ENDM
3440 ;
```

```
3450      .PAGE "    INPUT macro"
3460 ;
3470 ; MACRO:   INPUT
3480 ;
3490 ;  FORM:   INPUT ch,buf,len
3500 ;
3510 ; ch is given as in the @CH macro
3520 ; buf MUST be a proper buffer address
3530 ; len may be omitted, in which case 255 is assumed
3540 ;
3550 ; gets a line of text input to the given
3560 ;   buffer, maximum of length bytes
3570 ;
3580      .MACRO INPUT
3590        .IF %0<2 .OR %0>3
3600        .ERROR "INPUT: wrong number of parameters"
3610        .ELSE
3620          .IF %0=2
3630           @GP  %1,%2,255,CGTXTR
3640          .ELSE
3650           @GP  %1,%2,%3,CGTXTR
3660          .ENDIF
3670        .ENDIF
3680      .ENDM
3690      .PAGE "    CLOSE macro"
3700 ;
3710 ; MACRO:   CLOSE
3720 ;
3730 ;  FORM:   CLOSE ch
3740 ;
3750 ; ch is given as in the @CH macro
3760 ;
3770 ; closes channel ch
3780 ;
3790      .MACRO CLOSE
3800        .IF %0<>1
3810        .ERROR "CLOSE: wrong number of parameters"
3820        .ELSE
3830         @CH  %1
3840        LDA #CCLOSE
3850        STA ICCOM,X
3860        JSR CIO
3870        .ENDIF
3880      .ENDM
3890 ;
3900 ;;;;;;;;;;; END OF IOMAC.LIB ;;;;;;;;;;;
3910 ;
```

+

+

+

---this page intentionally left blank---

+

+

+

Appendix C: ERROR DESCRIPTIONS

When an error occurs, the system will print
*** ERROR -
followed by the error number (unless the error was generated with the .ERROR assembler directive) and, for most errors, a descriptive message about the error.

Note: The Assembler will print up to 3 errors per line.

The format used in the listing of descriptions which follows is simply ERROR NUMBER, ERROR MESSAGE, description and possible causes.

1 - MEMORY FULL
All user memory has been used. If issued by the Editor, no more source lines can be entered. If issued by the Assembler, no more labels or macros can be defined.
NOTE: If memory full occurs during assembly and the source code is located in memory, SAVE the source to disk, type NEW, and assemble from the disk instead. Now the assembler can use all of the space formerly occupied by your source for macro and symbol tables, etc.

2 - INVALID DELETE
Either the first line number is not present in memory, or the second line number is less than the first line number.

3 - BRANCH RANGE
A relative instruction references an address displacement greater than 129 or less than 126 from the current address.

4 - NOT Z-PAGE / IMMEDIATE MODE
An expression for indirect addressing or immediate addressing has resolved to a value greater than 255 ($FF).

5 - UNDEFINED
The Assembler has encountered a undefined label.

6 - EXPRESSION TOO COMPLEX
The Assembler's operator stack has overflowed. If you must use an expression as complex as the one which generated the error, try breaking it down using temporary SET labels (i.e., using ".=").

7 - DUPLICATE LABEL
The Assembler has encountered a label in the label column which has already been defined.

8 - BUFFER OVERFLOW
The Editor syntax buffer has overflowed. Shorten the input line.

9 - CONDITIONALS NESTING
The .IF-.ELSE-.ENDIF construct is not properly nested. Since MAC/65 cannot detect excess .ENDIFs, the problem must be an EXTRA .ELSE or .ENDIF instead.

10 - VALUE > 255
The result of an expression exceeded 255 when only one byte was needed and allowed.

11 - CONDITIONAL STACK
The .IF-.ELSE-.ENDIF nesting has gone past the number allowed. Conditionals may be nested a maximum of 14 levels.

12 - NESTED MACRO DEFINITION
The Assembler encountered a second .MACRO directive before the .ENDM directive. This error will abort assembly.

13 - OUT OF PHASE
The address generated in pass 2 for a label does not match the address generated in pass 1. A common cause of this error are foward referenced addresses. If using conditional assembly (with or without macros), this error can result from a .IF evaluating true during one pass and false during the other.

14 - *= EXPRESSION UNDEFINED
The program counter was forward referenced.

15 - SYNTAX OVERFLOW
The Editor is unable to syntax the source line. Simplify complex expressions or break the line into multiple lines.

16 - DUPLICATE MACRO NAME
An attempt was made to define more than one Macro with the same name. Only the first definition will be valid.

17 - LINE # > 65535
The Editor cannot accept line numbers greater than 65535.

18 - MISSING .ENDM
In a Macro definition, an EOF was reached before the corresponding .ENDM terminator. Macro definitions cannot cross file boundrys. This error will abort assembly.

19 - NO ORIGIN
The *= directive is missing from the program. Note: This error will only occur if the assembler is writing object code.

20 - NUM/REN OVERFLOW
On the REN or NUM command, the line number generated was greater than 65535. If REN issued the error, entering a valid REN will correct the problem. If NUM issued the error, the auto-numbering will be aborted.

21 - NESTED .INCLUDE
An included file cannot itself contain an .INCLUDE directive.

22 - LIST OVERFLOW
The list output buffer has exceeded 255 characters. Use smaller numbers in the .TAB directive.

23 - NOT SAVE FILE
An attempt was made to load or assemble a file not created with the SAVE command.

24 - LOAD TOO BIG
The load file cannot fit into memory.

25 - NOT BINARY SAVE
The file is not in a valid binary (memory image, assembler object, etc.) format.

27 - INVALID .SET
The first dcnum in a .SET specified a non-existant Assembler system parameter.

30 - UNDEFINED MACRO
The Assembler encountered a reference to a Macro which is not defined. Macros must first be defined before they can be expanded.

31 - MACRO NESTING
The maximum level of Macro nesting has exceeded 14 levels.

32 - BAD PARAMETER
In a Macro expansion, a reference was made to a nonexistent parameter, or the parameter number specified was greater than 63.

128 - 255 [operating system errors]
Error numbers over 127 are generated in the operating system. Refer to the OS/A+ manual for detailed descriptions of such errors and their causes.

a reference manual for

D D T

"Dunion's Debugging Tool"

a screen-oriented debugging program for
use with the OSS MAC/65 Macro-Assembler
on computers built by Atari, Inc.

Please contact Mr. Dunion or OSS, Inc., at
1221-B Kentwood Ave., San Jose, CA 95129
Telephone (408) 446-3099

Rev 1.Ø

PREFACE

DDT is the original design and product of Mr. James J. Dunion. Versions of DDT have been produced for disk based systems (sold through the Atari Program Exchange), but this version marks the first time DDT has been integrated with an assembler.

We at OSS like to think that it is especially appropriate that Mr. Dunion chose to allow us to link the fastest macro assembler for Atari computers with the most exciting concept in debugging tools. We hope you enjoy this powerful package as much as we have enjoyed preparing it for you.

TRADEMARKS

The following trademarked names may be used in various places within this manual, and credit is hereby given:

DOS XL, BASIC XL, MAC/65, and C/65 are trademarks of Optimized Systems Software, Inc.

Atari, Atari 400, Atari 800, Atari Home Computers, and Atari 850 Interface Module are trademarks of Atari, Inc., Sunnyvale, CA.

Table of Contents

Section 1: An Introduction to DDT

1.1 What DDT Is

The name "DDT" (a software analog to the biological bug killer of the same name?) has been used for many other debug programs on other systems (where it usually stands for "Dynamic Debugging Tool"). We at OSS are proud to offer the best and most "authentic" DDT, "Dunion's Debugging Tool", by Jim Dunion.

DDT has become one of the most popular debugging tools ever invented for use with Atari computers. In this version, OSS and Mr. Dunion have attempted to keep the spirit and flavor of DDT while scaling its size down enough to fit in an OSS SuperCartridge with MAC/65. This combination of MAC/65 and DDT is truly an all-in-one development system for assembly language programmers.

The heart of DDT is its ability to show what is happening inside the computer on a special display screen. This special screen is kept completely separate from your program's screen, whether you are using sophisticated graphics or simply Atari standard character I/O.

In effect, then, DDT tries to be as invisible as possible to your Atari computer's operating system, screen display handlers, keyboard handler. More importantly, though, DDT attempts to perform its tasks without interfering with your program.

This extraordinary separation of debugger and user program is coupled with the ability to easily change and monitor the internal state of "your" machine's environment, so that you can get a much clearer picture of exactly what's going on inside your system and program at any instant.

As with any software-based debugger, there are limitations on speed, instruction and memory tracing, and interrupt processing. All in all, though, DDT comes close to providing you with the best possible debugging environment, probably matched only by hardware logic analyzers costing hundreds of times more.

1.2 An Overview of the Workings of DDT

DDT is separated into four major functional parts: a display generator, a breakpoint handler, an instruction interpreter, and a user command processor.

Generally, when you enter DDT from MAC/65 (via the "DDT" command, of course), you are presented with an arbitrary display of a portion of memory with the values of the 65Ø2 registers (at the time DDT was entered). Naturally, if you intend to debug your own program, you must first tell DDT where it is. You do this via the command processor (but we won't discuss exactly how at this point).

If you are reasonably cautious, you will probably wish to step through your program a line at a time. You can do this thanks to DDT's instruction interpreter.

Once you have a subroutine or set of routines reasonably debugged through the use of single stepping, you will probably wish to execute them without full trace. Or you may wish to allow your program to run up to a certain point before you examine registers, memory locations, etc. DDT's breakpoint handler accomplishes both these tasks.

With a few exceptions, you accomplish all these tasks by using the command processor of DDT. By simple, easy to remember commands, you can ask DDT to interpret your program, show you the contents of memory in either instruction or memory dump formats, change memory or register values, and (in general) control the flow and environment of the program you are debugging.

1.3 An Example of Using DDT and MAC/65

We will present here a short and simple program, written in MAC/65, which we ask you to type in to the MAC/65 editor. We will then assemble and debug this program using DDT. We will not perform the more complex operations of DDT, but we hope that we will give you at least a feel for using DDT and its flexible commands.

```
+-----------------------------------------------------+
| NOTE: We assume here that you have read the MAC/65  |
|       manual and can use the MAC/65 editor and its  |
|       commands. For this example, though, we will   |
|       call out every keystroke to be used with DDT, |
|       including [RETURN] keys, unless we note       |
|       otherwise                                     |
+-----------------------------------------------------+
```

To begin, then, boot your DOS (if you are using a disk) and enter the MAC/65 cartridge. To the "EDIT" prompt, type "NUM" and enter the following program:

```
10 ; EQUATES -- FROM 'AMPPING THE ATARI'
20 HPOSP0 =   $D000     ;Hort. POSn, Player 0
30 PCOLR0 =   $02C0     ;Player COLoR 0
40 CHSET  =   $E000     ;addr of std char. set
50 PMBASE =   $D407     ;Player/Missile BASE addr
60 SDMCTL =   $022F     ;Set DMa ConTroL
70 GRACTL =   $D01D     ;GRAphics ConTroL
80 ;
90      *= $3800        ;an arbitrary address
0100 ;  SET UP FOR PM GRAPHICS
0110 ;
0120 SETUP
0130    LDA  # >CHSET   ;we use the char. set
0140    STA  PMBASE     ;...as data for player
0150    LDA  #4*16+4    ;color: hue 4, intensity 4
0160    STA  PCOLR0     ;for our player
0170    LDA  #$2A       ;std playfield,DMA,players
0180    STA  SDMCTL     ;...are all enabled
0190    LDA  #2         ;the bit for players
0200    STA  GRACTL     ;...is turned on
0210 ;
0220    LDX  #100       ;init our hort. pos'n
0230 ;
0240 LOOP
0250    STX  HPOSP0     ;where we want the player
0260    LDY  #10
0270 DELAY
0280    DEY             ;just wait for awhile
0290    BNE  DELAY
0300 ;
0310    INX             ;to next position
0320    JMP  LOOP
0330 ;
0340    .END
```

When you are satisfied that you have entered the program correctly, you might save it to disk or cassette and then assemble it. We used

```
ASM ,#P:
```

to get the listing which appears below. Of course, using the '#P:' requires that you have a printer hooked up to your computer, so you may wish to modify this command to suit your system's set-up (and see your MAC/65 manual for details on how to do so).

Verify that your listing is essentially identical (we have omitted the symbol table listing here).

```
                10 ; EQUATES -- FROM 'MAPPING THE ATARI'
    =D000       20 HPOSP0 =  $D000    ;Hort. POSn, Player 0
    =02C0       30 PCOLR0 =  $02C0    ;Player COLoR 0
    =E000       40 CHSET  =  $E000    ;addr of std char. set
    =D407       50 PMBASE =  $D407    ;Player/Missile BASE addr
    =022F       60 SDMCTL =  $022F    ;Set DMa ConTrol
    =D01D       70 GRACTL =  $D01D    ;GRAphics ConTroL
                80 ;
0000            90       *=  $3800    ;an arbitrary address
                0100 ; SET UP FOR PM GRAPHICS
                0110 ;
3800            0120 SETUP
3800 A9E0       0130     LDA  # >CHSET ;we use the char. set
3802 8D07D4     0140     STA  PMBASE   ;...as data for player
3805 A944       0150     LDA  #4*16+4  ;color: hue 4, intensity 4
3807 8DC002     0160     STA  PCOLR0   ;for our player
380A A92A       0170     LDA  #$2A     ;std playfield,DMA,players
380C 8D2F02     0180     STA  SDMCTL   ;...are all enabled
380F A902       0190     LDA  #2       ;the bit for players
3811 8D1DD0     0200     STA  GRACTL   ;...is turned on
                0210 ;
3814 A264       0220     LDX  #100     ;init our hort. pos'n
                0230 ;
3816            0240 LOOP
3816 8E00D0     0250     STX  HPOSP0   ;where we want the player
3819 A00A       0260     LDY  #10
381B            0270 DELAY
381B 88         0280     DEY           ;just wait for awhile
381C D0FD       0290     BNE  DELAY
                0300 ;
381E E8         0310     INX           ;to next position
381F 4C1638     0320     JMP  LOOP
                0330 ;
3822            0340     .END
```

Presuming that you have typed in and assembled this program correctly, it is time to lead you through the debugging process.

So give MAC/65 the "DDT" command, and you will be presented with a display similar to the one given below (though the screen version will be easier to read than our printed copy, thanks to inverse video, etc.).

```
+--------------------------------+
|    LOC. VAL INSTRUCTION        |
+--------------------------------+
|  DDT (c) 1984 JAMES J. DUNION  |
+--------------------------------+
|   >BED4  A9 LDA # $Ø4          |
|    BED5  Ø4                    |
|    BED6  48 PHA                |
|    BED7  2Ø JSR  $A5ØE         |
|    BED8  ØE                    |
|    BED9  A5                    |
|    BEDA  68 PLA                |
|    BEDB  38 SEC                |
|    BEDC  E9 SBC # $Ø1          |
|    BEDD  Ø1                    |
+--------------------------------+
|BKP1 BKP2 BKP3 BKP4 NV BDIZC    |
+--------------------------------+
|Ø000 Ø000 Ø000 Ø000 10110000    |
+--------------------------------+
| PC  A  X  Y  S  ENTER COMMAND  |
+--------------------------------+
|BED4 8D FF ØØ FF                |
+--------------------------------+
```

For now, let's not worry about what all that means. Suffice to say, DDT thinks that your program's PC is at location $BED4 and is showing you the code that it finds at that location.

But our assembly placed our main code at location $3800, so let's tell DDT to change what it is displaying. We do that by entering a command (which will be shown under the words "ENTER COMMAND") as follows:

```
        * 3800[RETURN]
```

NOTE that we do NOT type in the space between the '*' and the '3'. DDT does that for us.

Now look at the main display window. The '>' symbol should be pointing to location 3800. Do you see your code listed there? If you typed in the program exactly as we specified, and if you started from a "cold" (power-on) machine, you will probably find nothing but a series of 'BRK' instructions being displayed.

What went wrong? Actually, nothing. At this time, you should go back to MAC/65 by typing the DDT command 'Q' (just push the Q key, nothing else). Now type the following:

```
1    .OPT  OBJ
```

This line is necessary if you wish MAC/65 to assemble code and place the resultant object directly in memory. So, once again, you need to assemble your program. You may do so by simply typing

```
ASM
```

as a command to MAC/65. And, when the assembly is finished, you can go to DDT with the DDT command.

This time, after giving the

```
* 3800[RETURN]
```

command, you should see the beginning of your program displayed in DDT's main display window. Compare what you see to either your printed listing or the listing of Figure 1.2 to be sure that all is okay.

At last we are ready to try debugging our little program.

The first thing we will do is single step through the first part of our program. At this time, push the [OPTION] key one time. What happened? Presumably, the '>' is now pointing to location 3802. Also, the value of the PC (displayed under the letters 'PC') should be 3802. Notice expecially that the A-register now contains E0. In other words, we just executed the instruction 'LDA #$E0' which was at location 3800, and DDT is telling us what the new state of the CPU is.

Now push [OPTION] four more times, observing changes to the PC, display window, and A register. If you have done everything the same way we did, the A-register should contain 2A and the PC should be set at 380C. IF NOT, CHECK TO BE SURE YOUR PROGRAM MATCHES OURS!

Now comes the fun part. Push [OPTION] one more time. Did your display change dramatically? Remember, in section 1.1 we said there were a few limitations on display processing, etc.? We have just run into one of these limitations.

With this instruction (a STore A-register into SDMCTL, the system DMA control), we altered the width of the Atari's "playfield". DDT normally uses a narrow display. We requested a "normal" display. DDT accepts our choice and allows the change in display formats.

Surprisingly, DDT continues to function! And, if you are willing to ignore some of the junk on the screen, you can even read and understand most of the display. (Simply ignore the last 8 charactr positions on each line.)

We could "fix" the display (by pushing the [SELECT] button twice), but let us NOT do so at this time. (If we did, we wouldn't be able to see what happens next.)

Push the [OPTION] key four more times. Presto, an Atari "player" stripe full of character shapes appears. Since this is a demo of DDT, not an explanation of the Atari hardware characteristics, we don't want to spend too much time here explaining what has happened, but a very brief explanation will probably help you if now if you are not experienced with Atari hardware. The explanation which follows is given by address(es) from our little program.

3800-3804	By using the built-in character set as player 'data' we eliminate the to make player shapes for this demo.
3805-3809	This is the same as BASIC XL's PMCOLOR 0,4,0 and similar to SETCOLOR 0,4,0
380A-380E	We enable players and use a "standard" width playfield (character display)
380F-3813	This is a "must", to enable the player data registers. Actually, at this time the player is turned on and active. It's simply too far left of the screen to see.
3814-3818	Move the player stripe to horizontal position 100, which is a little left of the middle of the screen.

Now simply hold down the [OPTION] key. Watch the display of the registers. In particular, watch the values for the X and Y registers (displayed under the letters 'X' and 'Y'). Y seems to be decreasing at about one count per second. When it gets to zero, X is incremented and the player is moved right a little bit. Why? Because we stored X in the horizontal position register for our player.

If you continue to hold down [OPTION], the process will continue, albeit very slowly, and the player will move right across the screen. When you are tired of watching this, release the [OPTION] key.

Let's try something new. Push the 'I' key. What happened? Actually, what you are seeing is the same thing you saw when you held down the [OPTION] key, it's just happening much faster. You get to watch the registers changing, the instruction being executed moving (apparently up and down in the DEY loop, but that's an illusion), and the resultant movement of the player. Again, when you are tired of this, push the [BREAK] key.

So now we have seen two different speeds of instruction interpretation. But there is yet a third. First, though, push the [SELECT] key twice to restore DDT's normal display.

Again, enter the command sequence:

```
* 3800[RETURN]
```

And the PC and '>' displays should both again refer to location 3800. Push the [SELECT] key. The MAC/65 screen should reappear, just as you left it. CAUTION: you are NOT back in MAC/65! This simply demonstrates the independent screen display of DDT. Cute, yes?

Now, very carefully, push just the 'I' key. Once again, the player should appear and start moving across the screen. But now it is much, much faster. Why? Simply because DDT knows that it does not need to continually update its display of the registers, instructions, etc. Yet STILL your program is being interpreted!

When you are ready, press [BREAK] and DDT will regain control. For our last experiment, let's enter the DDT command sequence:

```
G 3800[RETURN]
```

Again, remember that DDT puts the space in for you. Do NOT type it in.

What happened? Presumably you have a very messy, smeared player moving impossibly fast across your display. This demonstrates the true speed of assembly language: the TV screen is not fast enough to keep up!

Push [CTRL][ESC] (hold down the [CTRL] key while pushing [ESC]). You should be back in DDT.

One final experiment: use the DDT command sequence:

```
E 381A[RETURN]
```

to move the display pointer '>' to location 381A. Then enter the sequence:

```
D 00[RETURN]
```

which alters the contents of 381A. Finally, again use the command:

```
G 3800[RETURN]
```

And observe the player, in more visible form, moving rapidly across the screen. Believe it or not, this is the slowest we can move the player if we use a simple single register delay loop (the code from 381B to 381D).

And now we are done with our demonstration. You may use [CTRL][ESC] to get back to DDT. Use 'Q' to return to MAC/65. Or simply reboot your system if you are done using DDT at this time.

Section 2: THE DDT SCREEN DISPLAY

The DDT Screen Display shows a user the internal state of the machine. The display screen is divided into several display areas which show different aspects of what is going on inside the computer.

Please refer to Figure 1.1 in the previous section for a rough picture of a typical display. Remember, to view the DDT display simply type the command 'DDT' from the editor of MAC/65.

The display areas are called :

- REGISTER DISPLAY -- Shows the current contents of the 65Ø2 registers
- DISPLAY WINDOW -- A window into memory
- BREAKPOINT TABLE -- Shows the settings of DDT's breakpoint registers
- COMMAND WINDOW -- Where you enter DDT commands from the keyboard

The following sections describe each of these display areas in more detail. However, for a full understanding of the capabilities of these deceptively simple displays, you must read this entire manual. And, of course, you should try using DDT. Only then will you understand how these displays can be used to their best advantage.

2.1 Register Display

The left side of the lowest part of the display screen is used to display the current contents of the 65Ø2 processor registers. Excepting that the status flag register is shown on the right side of the lines next to the bottom, on the same line as the breakpoints.

Whenever DDT is entered, the contents of the processor registers are copied into register shadows which are then displayed. These shadows are used to restore the 65Ø2 registers before control is released back to the program being tested.

In the next to last line of the DDT display, the names of the 65Ø2 registers are displayed. The current user-program values (contents) of these registers are shown (in hexadecimal notation) in the Register Display area directly beneath their names:

PC = Program counter
A = Accumulator
X = X index register
Y = Y index register
S = Stack pointer

Excepting for the PC, the values (contents) shown for these registers are all single byte values, thus displaying two hexadecimal digits. This is, of course, because all registers on the 65Ø2 CPU chip are a single byte in size. The sole exception is the Program Counter (PC), which is 16 bits (two bytes) in size and is displayed with four hexadecimal digits.

Not shown in the basic Register Display area is the processor status register. In order to allow you to more easily view and understand the value of the status register, it is shown in binary form. That is, each bit of the status register's contents is displayed in a special area of the DDT screen.

The legend "NV BDIZC" on the screen indicates that the bit values shown directly under the legend correspond to the various CPU status bits. In particular, the letters stand for (and the bit values are to be interpreted as):

N = Negative flag
V = Overflow flag
B = BRK instruction flag
D = Decimal mode flag
I = Interrupt disable flag
Z = Zero flag
C = Carry bit

The blank in the legend (and the corresponding bit under it) is an unused bit in the 65Ø2 status register and should be ignored.

2.2 Display Window

The display window forms a window into the system memory address space. This window is located in the top portion of the display screen, and occupies most of the screen. The window is set to an arbitrary address upon entry to DDT, but the initial address shown in the window may be changed by several commands (as described in later sections).

This display window may be thought of as having one of two possible filters in front of it.

The Disassembly Filter

The first filter, which is set upon initial entry to DDT is a disassembly filter. A GREATER THAN sign (>) points to what is called the current position.

In the disassembly display, each line from the current position down is shown in a similar format: the hexadecimal address of a location, its contents and then a disassembly readout. Standard 6502 mnemonics are used, with conventional address mode indications.

Note that the NCR 65C02 additional instructions and address modes are supported.

Several features have been added to aid debugging. If a mnemonic is shown in inverse video, it indicates that a breakpoint has been set at that location. In fact, if you look at the actual contents of that location, it will be a 0.

If the mnemonic in inverse video is a BRK instruction, that particular BRK instruction was not placed there by DDT. This would occur, for instance, in looking at memory that contains all zeros.

Secondly, if the instruction is one of the branch instructions, the computed target branch address is shown. An arrow (↑ or ↓) is used to indicate the direction of the conditional branch.

The Hexadecimal Filter

The second filter is a hexadecimal filter. This filter causes the display window to show the hexadecimal value and ATASCII representation of up to 40 memory locations. Again, the > sign indicates the current position.

If the hexadecimal filter is in place, each line after the current position line will start on an even 4 byte boundary.

This means the current position line can have 1 to 4 values on it. The current position line values will always be left justified.

2.3 Breakpoint Table

The Breakpoint table is located just above the register display.

There are four user definable breakpoints (labeled 'BKP1', BKP2', 'BKP3' and 'BKP4' in the display), each of which will be shown with its current setting.

If a register is clear (i.e., not set), then the value shown will be 0000.

If a breakpoint register is set, the value in that register will be the location (address) in memory where DDT has placed a BRK instruction.

2.4 Command Window

The extreme right hand part of the bottom of the screen is devoted to the command window. This is the area that shows the command that a user is typing in.

Often, a DDT command will consist of simply a single keystroke. Since DDT executes commands very quickly, you may never see the key appear in the command window. Be assured, however, that every key you type (other than the [OPTION], [SELECT], and [START] buttons) is echoed in this window.

Note that DDT commands requiring a following value, etc., automatically display a space after the first keystroke you type. This is for ease of understanding only. You do NOT type the space.

Section 3: An Overview of the DDT Commands

The command interpreter allows a user to issue keyboard commands to DDT. You may recall from Section 2 that the command window is shown in the lower right hand portion of the display screen.

Each DDT command requires only a single keystroke. If the key typed is not a valid DDT command, it will be ignored. If a key is a valid command and requires no additional arguments, the command which the key represents is executed immediately. Again, recall from Section 2 that most DDT commands execute so quickly that you may never see the command key echoed in the command window; but it really does go there, however briefly.

Some DDT commands, though, require one or more additional arguments. If you request a DDT command which needs one or more parameters, DDT will wait for you to enter the arguments it needs before proceeding.

SPECIAL NOTE: DDT always puts a space after the command key when it echoes the key in the command window. You do NOT type the space key. DDT places it there automatically.

COMMENT: In addition to the keyboard commands, DDT understands three "pushbutton commands", which are described in Section 5.

3.1 A Summary of the Keyboard Commands

The DDT Keyboard Commands are :

```
B <1,2,3,4>,<addr>..[1]  Breakpoint 1-4 set to given addr
D <hstring>.........[2]  Deposit hex string
E <addr>............[3]  Examine address addr
G <addr>............[4]  Go at address addr
I ..................[5]  Interpretive mode
M <addr><addr><len>.[6]  Move memory
N ..................[7]  Next instruction
Q ..................[8]  Quit, return to MAC XL
R <P,A,X,Y,S>,<val>.[9]  Register selected receives val
S <hstring>.........[10] Search for hex string
W ..................[11] Window filter toggle
↓ ..................[12] Move display window down/higher
↑ ..................[13] Move display window up/lower
* <addr>............[14] Set Program counter
```

In the list above, the numbers in square brackets (e.g., [3]) indicate the subsection number in chapter 4 where a full description of the command may be found.

The abbreviations enclosed in <angle brackets> are described in the LEGEND (in section 3.2), starting on the next page.

3.2 Legend

In the summary of section 3.1, certain abbreviations were enclosed in angle brackets (e.g., <addr>). In this section we explain the meanings and legal range of values for the data these abbreviations represent.

Also, these same abbreviations are used in Section 4, where each DDT command is described in detail.

You may recall that the abbreviations were as follows:
<1,2,3,4> <addr> <hstring> <val> <len> <P,A,X,Y,S>

We explain these abbreviations in two groups and then follow with some comments about delimiters.

3.2.1 Specific Selections: <1,2,3,4> and <P,A,X,Y,S>

When the commands 'B' or 'R' are used, each expects to be followed immediately by a single character. The characters between the angle brackets are the ONLY characters which will be accepted by DDT in each of these cases.

That is, if you type a 'B' as a DDT command, you MUST follow it with a '1', a '2', a '3', or a '4'. Any other characters are illegal.

If you type in the wrong character (e.g., you type 'B4' when you meant to type 'B3'), you may push the delete (back space) key. DDT will back up and delete the offending entry, and you may re-enter it.

See the descriptions of the 'B' and 'R' commands in Section 4 for more details.

3.2.2 Hexadecimal Values: <addr>,<val>,<hstring>,<len>

First, we must note that the abbreviations <addr>, <byte>, <hstring>, and <len> all represent hexadecimal values which you, the user, must type in. When DDT is expecting a hexadecimal value, it ONLY recognizes the characters Ø,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F (the traditional hexadecimal 'numeric' characters).

Specifically, DDT expects a certain number of hex digits, as noted in the following list:

<addr>	=	address value, 1 to 4 hexadecimal digits (i.e., 2 bytes)
<byte>	=	a single byte value, 1 or 2 hexadecimal digits
<hstring>	=	a hex string up to 12 digits long (i.e., 6 bytes)
<len>	=	a two byte length specification, must be either 3 or 4 hex digits

Generally, although DDT will accept fewer than the maximum number of digits, it will NOT accept MORE hex digits than it expects. Thus, if the legend <addr> appears in the summary of a DDT command, you will usually find that you will be unable to enter more than 4 characters (each of which must, of course, be a hex digit).

You can however, delete characters, and then enter new characters. Deleting back past the starting point of the value field will result in the previous item in the command being erased.

There are a couple of special cases in the above rules about field sizes, but they will be clearly described in Section 4, where individual commands are detailed.

3.2.3 Delimiters

There are two usages for delimiters.

First, the commands 'B' and 'R' require both a specific selection and a hexadecimal entry. You MUST separate the selection from the hex entry.

You may use either a [SPACE], a [COMMA], or a [RETURN] as a delimiter (separator). However, whichever you choose, DDT always DISPLAYS a comma as the delimiter.

Second, every hexadecimal value must terminated by a delimiter (except see Section 3.3 for the special cases of '*' and '>'). If DDT did not wait for such a delimiter, you would not be able to correct mistakes.

Again, you may use a [SPACE], a [COMMA], or a [RETURN]. Since a hexadecimal value is always the last item in the command entry, your delimiter is NOT displayed in the command window. Instead, the command is immediately executed.

Once a command has been executed, the command window is cleared to make room for your next command.

3.3 Special Characters: '*' and '>'

For input convenience, there are two special characters, '*' and '>'. These are used as shorthand ways of entering addresses (i.e., where the summary above calls for an <addr>).

'*' means the current value of the PC (as you might expect if you are familiar with 6502 assembly language). Generally, when an <addr> is called for, you may type just a single asterisk (*), and DDT will supply the current value of the PC (as displayed in the register display) for you.

Similarly, '>' means the current position of the Examine window pointer (the '>' symbol on the screen). Anytime an address is expected, you may type just a single greater than sign (>), and DDT will supply the address which the Examine window pointer (>) is pointing to.

In the command descriptions in Section 4, special note will be made if either or both of these characters are not legal for a given command.

SPECIAL NOTE: When either of these special characters is used as shorthand for an address, the command is immediately executed. DDT does NOT expect nor wait for a delimiter in this case.

CAUTION: Note that '*' is itself a legitimate DDT command. Do NOT confuse its usage as an address marker with its usage as a command.

Section 4: Command Descriptions

In this section, we present a more detailed description of each of the DDT keyboard commands. For the meaning and legal values of items enclosed in angle brackets (e.g., <addr>), please refer to Section 3.2. For usage of delimiters (shown in this section as commas), see Section 3.3.

The commands are presented in alphabetical order, as presented in the summary table in section 3.1.

4.1 B -- Set or Reset a Breakpoint

Format: B <1,2,3,4>,<addr>

Examples:
```
B 1,4000
B 1,,
B 2,*
B 4,>
```

You use the Breakpoint command to set (or reset) one of DDT's four breakpoint registers to a memory location (presumably an instruction byte) of your choice.

Note that two values (the breakpoint register number, and the breakpoint location) are required for this command. Both fields must be terminated with a delimiter.

To enter the command given in the first example, above, you could type 'B' then '1' then SPACE then '4000' then RETURN. (Remember, though, that all delimiters--SPACE, COMMA and RETURN--are treated identically. Remember, also, that DDT automatically supplies the space following the B. You do not type it in.)

If a value other than a 1,2,3, or 4 is entered for the breakpoint register, it will usually be ignored. If, however, you type in some other valid hexadecimal digit, the command will be terminated when you enter the following delimiter.

When a breakpoint is set, the location you specified shows up in the breakpoint register display under the breakpoint register number you specified.

If an Examine command is issued to look at a location in memory where a breakpoint has been set, a '00' data (instruction code) value will be seen, even though the proper mnemonic is shown in the disassembly.

Also, if a breakpoint is set at an examined location, the mnemonic will be shown in inverse video. This is a special feature of DDT, to make it easier for you to graphically see where a breakpoint is set and how.

If a breakpoint register is already in use when a new breakpoint is requested, the instruction at the old breakpoint location is first restored to its original value.

To clear a breakpoint register and restore the source code, type any delimiter after selecting the desired breakpoint register (e.g. typing 'B' then '1' then COMMA then COMMA will clear breakpoint 1 and restore the source code).

Trying to clear a breakpoint that is not set will not harm anything. Note, however, that trying to set a breakpoint in ROM, in hardware registers, or in non-existant RAM will have unpredictable (and possibly disastrous) results.

SPECIAL NOTE: Remember, you may use '*' and '>' as shorthand notations for the current value of your PC and the display window pointer. Thus you might examine memory until you find a location where you want a breakpoint. Then simply enter the command

```
B 2,> [RETURN]
```

(as an example only) to set breakpoint number two at the displayed location.

COMMENTARY: Physically, a '00' value (a BRK instruction) is stored in memory at the requested location. When DDT performs a disassembly and encounters a BRK instruction, it searches its breakpoint table to see if it had set that particular BRK. If so, it recovers the instruction for the disassembly but displays the mnemonic in inverse video.

4.2 D -- Deposit value(s) in memory

Format: D <hstring>

Examples: D Ø
D 313233343536
D 1234

The Deposit command is used to place one through six bytes in memory.

A string of hexadecimal values (up to 12 characters, 6 hex bytes) may be entered. The values entered will be placed in successive locations starting at the current position indicated in the display window (i.e., the address pointed to by the '>'), replacing whatever was there.

The input string is decoded two characters per hex byte at a time. If there is an odd character left at the end, it will be interpreted as the low order nibble of a hex value.

For example, entering a string of ØlAABØ will result in three bytes (Øl, AA, and BØ) being placed in memory. However, entering ØlAAB will result in Øl, AA, and ØB being deposited.

Note that depositing a byte or a series of bytes will NOT move the display window. This must be done with the examine or the move window up or down commands.

SPECIAL FEATURE !!

DDT is able to switch screens by saving 13 locations the operating system uses in managing the system graphics. Thus, before each value is deposited, it is examined to see if it should be deposited to these graphics locations. If so, the value is placed instead in an internal save table. Thus, for example, you can deposit values directly to the color shadow registers and affect the color of the user screen and not the DDT screen.

See Section 8.5 for a list of the locations saved in this fashion.

4.3 E -- Examine memory

Format: E <addr>

Examples: E 5000
E *
E 0

The Examine command is used to set the display window to view an area of memory. The extreme left hand edge of the display window has a GREATER THAN sign (>) in the 3rd row. This points to what we refer to as the "current position" in the display window.

Unless you have used the '↓' or '↑' commands, the current position will be the address entered via the last 'E' command.

Note that the 'E' command does NOT change the state of the display window filter, nor will it affect which instruction will next be executed by a single step command.

Since you may specify any arbitrary address as the location to be Examined, and (if you are using the disassembly filter) since you may accidentally disassemble a nonsense intruction byte, we recommend one or more of the following:

1. Examine only locations known to contain valid instruction bytes. Refer to a printer listing to be sure you are doing so.

2. After using 'E', move the display window up (lower in memory) a few bytes and then back down (via the '↑' and '↓' commands), to ensure that you are displaying instructions which are on true instruction boundaries.

3. Examine a few bytes ahead of where you really want to be. Then move down (via the '↓' command) to the proper position.

(See also the SPECIAL NOTE in Section 4.2.)

4.4 G -- Go to a Program at a Given Address

Format: G <addr>

Examples:
```
G 5000
G *
G >
```

The Go command is used to begin execution of your program at a specific location in memory.

Before control is transferred to this location, several actions take place:

1. All registers are updated based upon the current contents of the displayed registers.

2. The 13 locations saved for the graphics display (see Section 4.2, above, and Section 8.5) are restored, thus restoring your display and removing DDT's display from the screen.

3. Vertical Blank Interrupts and Display List Interrupts are BOTH enabled.

Obviously, since Going to your program can be dangerous (e.g., your program may wipe out all of memory, attempt to illegal I/O, or other miscellaneous nasties). We therefore urge caution on your part (including, at the least, saving your latest version of your program to disk or cassette) before using this command.

For all intents and purposes, once you issue a Go command your program has complete control of the Atari computer. There are two methods of returning to DDT: (1) If your program executes a BRK instruction (a zero instruction byte), DDT is entered at its breakpoint entry (see Section 7.3). (2) If you push [CTRL][ESC] (hold down the [CTRL] key while hitting [ESC]), DDT is entered at its "flash" entry point (see Section 7.2).

Method 1 is the most common method and is commonly used when debugging. Method 2 is an emergency method, reserved for when your program starts looping and nothing else will get you out.

Breakpoints are discussed in some detail in Section 6. The "Flash" entry point to DDT is discussed in Section 7.2.

NOTE: The special command sequence 'G *' is exactly equivalent to pushing the [START] button. See Section 5.1 for usage of the [START] button.

4.5 I -- Interpretive Mode

Format: I

Example: I

The Interpretive Mode command is used to place DDT in an automatic single step mode.

Interpretive mode will run with either the user screen or the DDT screen being shown, but you pay a severe time penalty for selecting the DDT screen. After each instruction is interpreted, the screen display is updated if the DDT screen is turned on. The display window is automatically placed in the disassembly mode, and all registers are displayed along with the updated disassembly.

Interpretive mode runs much faster if the user screen is selected, because DDT does not have to update it's screen if it is not active. See Section 5.2 for information on how to enable and disable your display screen when using DDT.

Pressing the BREAK key halts the interpretive mode. Encountering and attempting to execute a BRK instruction halts the interpretive mode.

COMMENTARY: When in interpretive mode, DDT attempts to execute your program as true to form as possible. To this end, DDT moves the instruction pointed to by your PC to a special working area and executes it at that location. Although, if the instruction is one which transfers control (e.g., JMP, JSR, BEQ, etc.), DDT truly "interprets" it.

Also, before DDT executes each instruction, it restores all your registers to the values shown in the register display. After executing (or interpreting) the instruction, DDT restores the proper register values in the register display.

SPECIAL NOTE

Because of the way interpretive mode works, you MAY interpret through ROM-based code. You should NOT, however, attempt to interpret any real-time I/O code (whether in ROM or not), including disk and other serial I/O.

4.6 M -- Move memory

Format: M <addr><addr><len>

Examples: M E000600000400
M 600060010040

The Move memory command simply does what its name implies: it moves one or more bytes of memory from one location to another.

This command requires a somewhat special format for its values. Specifically, all three values (both <addr>'s and the <len>) MUST be given, but you are NOT allowed to put ANY delimiter(s) (including spaces) between the values.

Both the <addr> values MUST be specified with EXACTLY four hexadecimal digits (using leading zeroes if needed).

The <len> may be any number from 0001 to FFFF (though disastrous results will obviously occur if you try to move all--or even major significant portions--of memory), but even <len> must be specified with three or four hexadecimal digits.

The first <addr> given is assumed to be the source or "from" address. The second <addr> is thus the destination or "to" address. And, of course, the <len> specifies the number of bytes to move.

Thus, the first example shown above will move $0400 (1024 decimal) bytes from memory location $E000 (through $E3FF--the main character set area of ROM) to memory location $6000 (through $63FF).

DDT does NOT check for possibility of overlapping "from" and "to" memory areas before it does the move, so an attempt to use a Move as in the second example above may or may not work the way you expect it to.

4.7 N -- Next

Format: N

Example: N

The Next command is really a shorthand method of program tracing which combines some of the best features of breakpoints with the ease of interpretive mode.

Using the Next command is equivalent to visually examining the disassembly display, determining the address of the next instruction (after the one the '*' is pointing to), setting a breakpoint at that address, and (finally) executing a 'G *' command (or [START] pushbutton command--see Section 5.1).

Most of the time, then, using N is equivalent to interpreting a single instruction (as may be done via the [OPTION] button--see Section 5.3). However, there are several important differences:

1. The Next command uses its own internal breakpoint and places it after the next instruction to be executed. This internal breakpoint is never displayed.

2. The user's screen is restored (as with the Go command, Section 4.4, above) while the instruction is being executed.

3. The instruction is truly executed, not interpreted, so you may not use 'N' when your PC points to ROM code.

4. If the instruction being pointed to by your PC (the '*') is a JSR, then the entire subroutine will be executed before DDT regains control! This allows you to execute ROM code or real-time I/O code at full processor speed and yet view the results immediately after the called routine finishes.

   CAUTION: If your subroutine performs an error exit and does not "properly" return (presumably via an RTS instruction) to the calling program, the breakpoint set by 'N' may never be executed.

5. If the instruction being pointed to by your PC is a JMP or branch instruction, you should usually NOT use the 'N' command, since the program may never reach the point where the internal breakpoint has been set.

4.8 Q -- Quit DDT, Reenter MAC/65

Format: Q

Example: Q

There is nothing fancy about this command. It is simply a means of exiting from DDT back to MAC/65.

Before transferring .control to MAC/65, DDT restores MAC/65's zero page locations and its critical page 4 locations (as described in Section 8.1).

DDT also removes all its own breakpoints from user code before Quitting and "unhooks" its Flash entry point from the system keyboard routine (see Section 7.2 and 8.2).

Upon re-entry to DDT (via MAC/65's "DDT" command), the user should restore any critical breakpoints by hand.

4.9 R -- Register Modify Command

Format: R <P,A,X,Y,S>,<val>

Examples:
```
R A,ØØ
R X,FF
R P,Ø1
```

The Register command is used to modify the contents of any of the 65Ø2's registers except the PC.

After typing 'R', only a 'P','A','X','Y', or 'S' will be allowed. Any other character will be ignored. No other character other than [DELETE] will be allowed until a delimiter is typed.

'P' indicates the processor status register (which is displayed in binary form under the "NV BDIZC"). 'A', 'X', and 'Y' are the normal 65Ø2 registers of the same names. 'S' represents the value of the stack pointer.

After entering the register designator , only two hex digits (i.e. one byte) will be accepted. Note that this command requires two separate values and two separate delimiters.

WARNING! Indiscriminate use of this command to change the stack value (the 'S' register) could make it impossible for DDT to continue to function without being reset.

4.10 S -- Search for a String of Bytes

Format: S <hstring>

Examples:
```
S 31
S 5F5F
S 8D0003
```

The Search command is used to locate a specific sequence of bytes in memory.

You may enter a hex string of up to 12 characters which will be interpreted as up to 6 bytes. DDT will search for the string you specify, starting from the current position (as indicated by the '>' in the display window) upwards (increasing addresses) through memory.

If the search is successful (the sequence of bytes is found), the display window will be repositioned (and the '>' will point to the first byte of the found sequence). If it is unsuccessful, the command window will simply be cleared for the next command, and the display window will not move.

If no value is entered after the 'S' (i.e. just a delimiter is typed), the previous search string will be used. This allows for easily finding multiple occurences of the search string.

The three examples given above might be interpreted as follows:

```
S 31  -- find a '1' character
S 5F5F   -- find a pair of question marks ('??')
S 8D0003 -- find a 'STA $0300' instruction
```

4.11 W -- Window Change Command

Format: W

Example: W

The Window command is used to change the "filter" over the display window.

You will recall from Section 2.% that there are two different "filters" available to you: a disassembly filter and a hexadecimal filter.

The 'W' command simply toggles between the two.

Note that certain commands will automatically change the filter to their "desired" state. You may use the 'W' command to change the filter back to the one you wanted if your choice does not correspond to DDT's.

4.12 ↓ -- Move Display Window Down (Higher in Memory)

Format: ↓

Example: ↓

The Move Window Down command is used to change the memory being displayed in the display window.

Specifically, the '>' pointer will be changed to point to a location higher in memory. How far the window and pointer are moved depend on which filter (hexadecimal or disassembly) is in place at the time the key is pushed.

If the hexadecimal filter is in place, pushing the '↓' key will move the window down (higher in memory) by one byte.

If the disassembly filter is in place, pushing the '↓' key will move the window down (higher in memory) by one full instruction (which may be one, two, or three bytes).

SPECIAL NOTE: You should NOT hold down the CTRL (control) key when using this command. DDT recognizes '=' as the 'down arrow key' even without CTRL pressed.

ALSO NOTE: Auto Repeat on the keyboard IS active, so that continuing to press the '↓' key will continue to move the window down.

4.13 ↑ -- Move Display Window Up (Lower in Memory)

Format: ↑

Example: ↑

The Move Window Up command is used to change the memory being displayed in the display window.

Specifically, the '>' pointer will be changed to point to a location one byte lower in memory.

Since an instruction could be 1,2 or 3 bytes long, you must be careful to watch and ensure that you remain on instruction boundaries if the disassembly filter is in place.

SPECIAL NOTE: As with the '↓' key (section 4.12, above), you should NOT use the CTRL key to select '↑' and auto repeat IS active for '↑'.

4.14 * -- Set Program Counter

Format: * <addr>

Examples:
```
* 5000
* >
*
```

The command is used to set the program counter.

After you enter the '*' command, DDT expects you to enter an address which will become the new PC contents.

After changing the PC, you may use the 'I' or 'N' commands or the [OPTION] or [START] buttons to begin or continue program execution (or interpretation) at the new location shown in the PC portion of the register display.

DDT always selects the disassembly filter after executing the '*' command and always sets the display window pointer (>) to the same address as the PC.

Note that you may type '*>' as a shorthand notation to set the PC to the address currently being shown in the display window (as indicated by the '>' pointer).

Note also that you may simply type '*' followed by [RETURN] to force the display filter to hexadecimal and force the display window pointer equal to the PC. This can be thought of as a shorthand notation for 'E*' (see Section 4.3) possibly followed by 'W' (see Section 4.11).

Section 5: Push Button Controls

The three ATARI console push buttons are used by DDT for useful and special operations. In many ways, you may think of these butons as extensions to the commands given in Section 4.

Each console button has a unique use, which is described below.

5.1 The START Button

A press of the START button is usually indicated in this manual by the notation [START].

[START] is used to continue code execution at the location indicated by the PC register.

Your screen display is restored and all 6502 registers are updated with the current displayed contents before control is transferred.

Pushing [START] is functionally equivalent to executing the command sequence 'G*', and we suggest reading Section 4.4 for more information on the Go command.

5.2 The SELECT Button

A press of the SELECT button is usually indicated in this manual by the notation [SELECT].

[SELECT] is used to toggle back and forth between the DDT screen and whatever screen dynamics were active before DDT was called and/or reentered (e.g., via a breakpoint).

An attempt has been made to allow for most alternative display features such as mixed Display lists, VBLANK routines, alternative character sets, display list interrupts, playfield size changes, and player-missiles. Thus, whenever DDT is entered or reentered, the locations necessary to restore these features are "remembered" by DDT before DDT puts its own display on the screen. When you execute your program (via the 'G' command, the [START] button, or the 'N' command), DDT restores your screen display as well as it can (and it usually does pretty well).

Generally, then, [SELECT] has only two primary purposes:

1. When you simply wish to look at your display screen momentarily.

2. When you wish to interpret your program (via the 'I' command or the [OPTION] button) while keeping your display active instead of DDT's.

5.3 The OPTION Button

A press of the OPTION button is usually indicated in this manual by the notation [OPTION].

[OPTION] is used to "single step" the processor through your program.

This causes the disassembly filter to be turned on, but will not automatically toggle the display screen. Holding down the OPTION button will continue to single step, at the rate of approximately two instructions per second.

Excepting for the fact that only a single instruction is executed before a pause is made, the [OPTION] button "command" works identically with the 'I' (Interpretive Mode) capability. Therefore, see Section 4.5 for more details on interpreting code via DDT.

Note that you may NOT interpret a BRK instruction. The interpreter will, for all intents and purposes, halt when it encounters a BRK.

Section 6: Breakpoints and Breakpoint Processing

One of the most common debugging techniques is to make use of a breakpoint.

This manual contains much additional information on breakpoints, so we refer you also to Sections 7.3, 4.1, 4.7, and 8.4. This Section will attempt to provide an overview on breakpoints as well as suggested uses for them.

The fundamental mechanism of a breakpoint is fairly simple:

1. When a running program encounters a 'BRK' instruction (a zero byte), the 6502 CPU simulates an interrupt (an IRQ, not an NMI, except that the 'SEI' instruction can NOT disable 'BRK' interrupts).

2. The only real difference between a true IRQ and a BRK-simulated interrupt is that a BRK causes the 'B' bit (bit 4, $10) to be set upon entry to the interrupt handler.

3. When the BRK-simulated interrupt occurs, Atari's OS uses the 'B' bit to recognize the fact and transfers control, via a RAM vector, to DDT.

4. DDT's breakpoint entry simply saves all the user's registers (A,X,Y,Processor status, Stack, and the Program Counter). It then sets the display window pointer (>) equal to the user's PC, selects the disassembly filter, saves the usual graphics information (see Section 4.2 and 8.5), and presents you with the typical DDT screen display.

After a breakpoint has been encountered, and control has been transferred to DDT, there are several ways to leave DDT. The 'N' command (Section 4.7) will set a breakpoint at the next location and then continue code execution. [START] (section 5.1) simply continues code execution. 'G' (section 4.4) can be used to transfer control to another location.

There are three ways to set a BRK instruction and thereby allow a breakpoint to happen.

1. You can use the 'B' command (as described in Section 4.1) to set up to four special DDT breakpoints. There are two advantages to this method: (a) DDT remembers the instruction which was at the location before you set the breakpoint, so when you reset or remove the breakpoint DDT can automatically restore the instruction for you. (b) The disassembly display shows your original instruction in inverse video, as a convenient reminder.

2. You can actually store a zero byte (a BRK instruction) in your code. You can do this either with the 'D' (Deposit) command or by actually assembling a BRK in you source code.

3. You can use the 'N' command (Section 4.7), which automatically sets a BRK instruction in the byte which follows the current instruction. Again, as with the 'B' command, DDT remembers your original instruction and restores it without effort on your part. Note that you will never see the BRK placed by 'N', as it is automatically removed as soon as DDT recovers control.

The best use of multiple breakpoints is to set one at every path in your program where you do NOT expect to or want to go (execute). That way, if your program takes a wrong turn, DDT will alert you by saying, "Hey! How'd we get to this breakpoint?"

Also, of course, you will normally step through your code a little at a time, setting a breakpoint a little farther ahead each time. For this use, we recommend reserving a single breakpoint register (usually number 1). Use the other registers (2 through 4) for the "side" or unexpected paths mentioned in the previous paragraph.

When one of the breakpoints is encountered in intrepretive mode, it will halt the intrepretive mode at that point.

Section 7: DDT Entry Points

There are four ways of entering or reentering DDT:

MAIN ENTRY
FLASH ENTRY
BREAKPOINT ENTRY
RESET ENTRY

Each is described separately below.

Sometimes, it will seem that the computer has locked up and none of the Entry methods described below will work. Generally, this is because something has gone wrong in the program you are debugging, and it has modified certain critical memory locations.

Disabling interrupts (executing an 'SEI' instruction) and/or modifying the interrupt vectors of Atari's OS are particularly insidious ways of destroying DDT's access to the system. And accidentally using the Move command incorrectly can obviously wipe out wholesale hunks of memory.

These are obviously only some of the ways to effectively disable DDT, but we would hope the most users will not encounter any of them. It is usually only the more sophisticated and complicated programs which will be altering locations which DDT is sensitive to.

7.1 Main Entry to DDT

When you give MAC/65's editor the "DDT" command, DDT is entered at what we call its Main Entry point.

Section 8.1 describes in some detail the process DDT goes through when it is entered. In particular, DDT saves the state of MAC/65 so that you do not lose your source code.

See also Section 8.1.

7.2 "Flash" Entry to DDT

This entry point is provided to allow immediate reentry to DDT regardless of most other circumstances.

When DDT is called, the operating system code that looks at the keyboard is modified so that it looks for a special character first, before handling normal keyboard input.

The special character looked for is one which is unused by normal Atari operations: [CTRL] [ESC]

In other words, to reenter DDT when your program is running, simply press both the Control and the Escape keys at the same time.

When DDT's modified keyboard handler finds the [CTRL][ESC] character, DDT is entered immediately through the FLASH ENTRY point (which is essentially equivalent to encountering a breakpoint).

Using the 'N' command or pressing START will return control to whereever the processor was at when the DDT special character was typed.

For more information on the Flash entry mechanism, and some warnings about how you may inadvertantly make it inoperative, see section 8.2.

7.3 Breakpoint Entry to DDT

Breakpoint entries are the most common way of entering DDT.

Once DDT has been entered via the Main entry, DDT's breakpoint handler is set up. Thereafter, anytime your program (or, for that matter, any program) attempts to execute a BRK instruction (a zero byte), DDT is entered at its Breakpoint Entry.

For more information on the use and characteristics of breakpoints, see Sections 6 and 4.1.

7.4 RESET Entry to DDT

If DDT was active before you executed your program (e.g., via the 'G' or 'N' commands or the [START] button), then pushing the [RESET] button should return control to DDT.

Obviously, if your program has scrambled enough locations which are vital to DDT and/or the Atari OS, then the RESET handling may never have a chance to occur.

Section 8: Technical Details of DDT

8.1 Interaction with MAC/65

DDT is designed to be compatible with MAC/65 so that you can easily go from editing to assembling to debugging and so forth.

Specifically, you enter DDT via the 'DDT' command from the MAC/65 editor. At that point, DDT saves certain memory areas which are critical to MAC/65's functioning in memory reserved by MAC/65 (and pointed to by MAC's 'lomem' pointer--the first value given in the response to a 'SIZE' command in MAC).

When you use the 'Q' command to exit DDT and reenter MAC/65, DDT restores the critical memory areas. If, during the course of your debugging session with DDT, you have not changed any of the memory bounded by the low and high values given in response to MAC's 'SIZE' command, you will find your source code (if any) intact and ready to edit and/or (re)assemble.

MAC/65 and DDT cooperate in another way: when you push the [RESET] button on the Atari keyboard (hopefully, only as a last gasp desparate measure), MAC/65 attempts to determine whether DDT or MAC had control when the button was pushed. If DDT had control, MAC automatically and immediately reenters DDT at a special RESET entry point.

8.2 Keyboard Scanner

During DDT initialization the system keyboard vector is redirected to a preprocessor which checks for the DDT FLASH ENTRY special character ([CTRL] [ESC]). If this character is seen, control transfers to the FLASH ENTRY point, otherwise control passes to the normal keyboard processing routine.

Note that keyboard interrupts must be enabled. If your program alters or disables the keyboard interrupt (or its vector), DDT will not be able to regain control. You may or may not be able to push [RESET] to reenter DDT.

Not that this implies that the 'SEI' instruction will also disable the DDT keyboard scanner. This is somewhat important, but since 'SEI' disables all keyboard activity we would hope it is an instruction you will use with care.

8.3 DDT's use of System Resources (RAM and ROM)

DDT itself occupies a portion of the MAC/65 cartridge space. When it is called, the upper half of page zero and certain locations in page four ($400-$4FF, but not all of this range) is saved for later use by MAC/65 (see 8.1, above).

While DDT is running, then, locations $80 through $AF are used by DDT and should be avoided by user programs. Otherwise, the locations in the upper part of page zero may be used.

Also, DDT takes the RAM area from $3FD through $57F for its display screen, breakpoint registers, etc. Since the cassette buffer occupies $3FD to $47F, this implies that you can NOT do cassette I/O from within DDT (though you may load a tape from MAC/65 before entering DDT or save to a tape after exiting).

Remember, also, that MAC/65 is NOT capable of assembling directly into page six ($600 through $6FF). You may, however, assemble into other (higher and safer) memory with an offset (see the MAC/65 manual, Section %.%%) and then use DDT's Move command to place the resultant code in page six.

8.4 Things to Watch Out For

There are a few areas where you have to be careful in using the DDT cartridge. In general, these occur when you are single stepping or running interpretively.

If the code being interpreted alters the display list or does direct access to ANTIC or CTIA/GTIA, then you might end up with a scrambled screen. Usually this is non fatal, just distracting. (See our example program and debug session in Section 1 for an instance of just this occurrence.)

To restore the normal DDT screen, press the BREAK key to halt the interpretive mode, then press SELECT twice (though doing so may turn off any players, etc., which you had made active).

Trying to do I/O from disk or any other real time activity in either interpretive mode or single step mode will almost certainly not work.

You should set up breakpoints around the real-time code so that this type of I/O is done in real time. For example, try using the "N" command anytime your code does a JSR to CIO or SIO.

8.5 Graphics Location Saved by DDT

Whenever DDT is entered (see section 7), it saves certain memory locations pertaining to user graphics before presenting its own display (also see Section 4.2). The locations saved are all "shadow registers" or vectors in page two. The following are a list of the locations saved, by label, hex address, number of bytes saved, and very brief description. The labels given are those used in "Mapping the Atari" (from Compute! Books) and in the Atari OS listings (part of the Atari Technical Manual set), and we refer you to those publications for more information.

Label	Address	# bytes	Description
VVBLKI	$Ø222	2	VBI immediate vector address
VVBLKD	$Ø224	2	VBI deferred vector address
SDLSTL	$Ø23Ø	2	Start address of display list
SDMCTL	$Ø22F	1	DMA control register
GPRIOR	$Ø26F	1	Priority selection register
COLOR1	$Ø2C5	1	Color register 1
COLOR2	$Ø2C6	1	Color register 2
COLOR4	$Ø2C8	1	Color register 4
CHACT	$Ø2F3	1	Character mode register
CHBAS	$Ø2F4	1	Character set base address

If your program uses other system memory locations which are altered by DDT, or if your program changes graphics characteristics by direct changes to the Atari hardware registers, DDT will NOT be able to completely restore the screen display your program was exhibiting when DDT was entered.

8.6 Using MAC/65 as a Mini-Assembler for DDT

Those of you who have used other debugging tools may note that, while DDT has a fairly sophisticated disassembler, it lacks a built-in mini-assembler. Thanks to the integrated nature of MAC/65 and DDT, though, you may never even notice this omission.

Let us suppose, for the moment, that you have just assembled a small to medium sized program from source code in memory, placed the object code in memory, and have entered DDT. When you discover an error in your code, you can simply pop back to MAC/65, change the offending code, re-assemble, and be back in DDT in a matter of a very few seconds.

But what if the code you want to patch is NOT in memory or is not directly related to the source currently in memory. What can you do then? We suggest the following steps:

```
Exit DDT via the Q (Quit) command.
If there is a source program in MAC/65's edit
        buffer, type in  "RENUM 1000,1"
Type NUM 10,10.
Enter your patch, using "*=" to control where
        the patch goes and " .OPT OBJ" to
        ensure the patch really ends up in memory
Assemble via the "ASM" command.
Go back to DDT (via "DDT", of course), and your
        patch is in place.
NOTE: to get rid of your patch code without
        affecting your main program, you may
        type "DEL 1,999" to MAC/65.
```

There ARE some things to watch out for if you use this method. Primarily, you want to ensure that MAC/65 doesn't wipe out the program you are trying to debug. There are two possible ways this could happen.

First, remember that MAC/65 destroys the lower half of page 6 ($600 through $67F). If you are using page six, then, you should use the Move command to move page six to someplace "safe" before going to MAC/65 (then move it back when you return to DDT).

Second, the very process of editing (writing) even a small program will overwrite some of memory. However, we direct your attention to the MAC/65 LOMEM and SIZE commands. You can use LOMEM to set the bottom of the memory that MAC/65 will use. You can use SIZE to determine what memory MAC/65 is, indeed, using. We suggest, if you intend to use the method we have outlined, that you use LOMEM when you first enter MAC/65.

The other major problem you can encounter relates to the way DDT saves the state of MAC/65 when it is entered. Since the two programs (and they really do operate as almost totally separated programs) share some of the same memory space, DDT saves part of page zero and part of page four ($80-$FF, $480-$4FF) when it is entered. It saves these locations at the start of MAC/65's "buffer" space.

You can determine where the buffer space is by using SIZE: the first number displayed in response to SIZE is the hexadecimal address of this buffer. You can change where this buffer is by using LOMEM.